Aalto University
School of Science
Master's Programme ICT Innovation

Timon Schneider

Intelligence Service Schema:

A Semantic Metadata Model for Service Discovery in the Intelligence Layer

Master's Thesis
Espoo, June 15, 2020

Supervisors: Professor Eero, Hyvönen
Advisor: Edgar, Ramos M.Sc. (Tech.)

Aalto University
School of Science

Aalto University
School of Science

Master's Programme ICT Innovation

ABSTRACT OF
MASTER'S THESIS

Author:	Timon Schneider		
Title:	Intelligence Service Schema: A Semantic Metadata Model for Service Discovery in the Intelligence Layer		
Date:	June 15, 2020	**Pages:**	59
Major:	Data Science	**Code:**	SCI3095
Supervisors:	Professor Eero, Hyvönen		
Advisor:	Edgar, Ramos M.Sc. (Tech.)		

The problem of engrained intelligence functionality in Internet of Things (IoT) applications is discussed. Offering intelligence services to applications through a separate layer is discussed as an alternative. Two important advantages are improved life-cycle management and increased potential of value sharing. A solution that allows device owners to install intelligence services that can be used by local applications is suggested. This thesis explores discovery process and the representation of such intelligence services through the use of a metadata schema, the Intelligence Service Schema. This schema allows for semantic description of intelligence services which makes advanced service discovery possible through the semantic query language SPARQL.

Keywords:	Intelligence Service Discovery, Semantics, AI Distribution, Intelligence Description, Intelligence Market
Language:	English

Acknowledgements

I want to thank Edgar Ramos for the opportunity and guidance. It was a pleasure to work at Nomadic Lab Ericsson!

Doorwerth, June 15, 2020

Timon Schneider

Contents

Chapter 1

Introduction Statement

Intelligence is considered one of the most important functionalities of Internet of things devices and is already used in many IoT solutions [16]. Experts estimate that by 2020, 15 to 30 billion connected devices will be in use [16]. The expected production of data is anywhere from 44 to 600 zettabytes per year [19, 32]. Efficiently capturing the value of this enormous amount of data requires machines instead of humans. Automatic processing and machine intelligence functions like optimization, anomaly detection, prediction, and error correction can be applied to autonomously derive knowledge from the data, report, or optimize the operations of the device without human interference.

As of right now, the deployment of such machine intelligence functionality is done in a centralized manner. Execution of the algorithms and models often takes place in remote or local data centers. Either way, the algorithms and models are often trained centrally with data reused for multiple machine intelligence implementations leading to dataset bias [43] and a limited ability to customize and control the devices' machine intelligence functionality. There are also other reasons to have intelligence functionality within the device. In certain situations it is preferred or necessary that the data is processed locally. The data might be too sensitive to transfer over the web and is required to stay on the device. Another reason could be that the device is expected to keep full functionality when it loses network connection.

When the intelligence functionality is executed locally, it is often highly engrained in the application layer. This works well if the systems and applications are tightly integrated, the boundaries are well defined, the environment is not expected to change, the intelligence functionality remains the same, and the task and goal do not change. For environments where the need for intelligence functionality is susceptible to change, the limitations of integrating intelligence into the application are obvious; once the intelligence

task changes, the whole application needs to be updated. This leads to a significant amount of overhead work.

Decoupling intelligence from the application layer, as shown in Figure 1.1, can addresses this problem by making the intelligence as independent of the application as possible. Now, the intelligence becomes a separated stratum that provides services to applications in the same way that other layers and platforms provide services [38]. In this way, intelligence functionality can be changed, updated or upgraded without the need of changing the application.

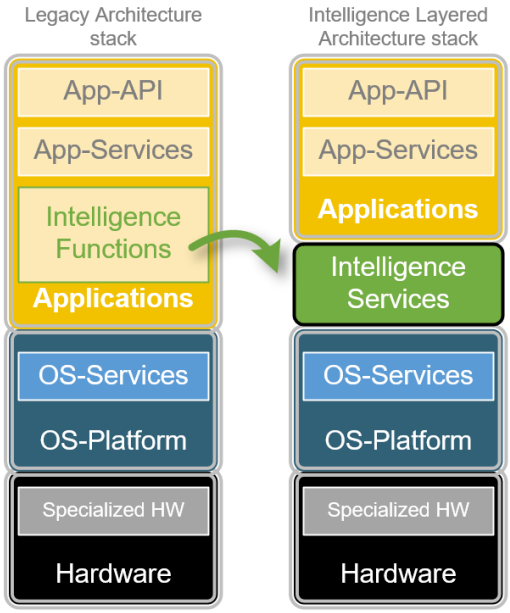

Figure 1.1: Layering of legacy architecture stack and layering of intelligence layered architecture stack. Adapted from [38].

The decoupling of intelligence functionality from the application layer is a relatively new concept. There are many aspects of the architecture that need further research and testing. Figure 1.2 is an overview of an ecosystem that would enable the decoupling of intelligence services in devices while having full intelligence functionality locally available on the device. The intelligence broker stores intelligence services uploaded by intelligence providers. Applications on a device would request intelligence functionality from an intelligence layer that hosts intelligence services. The intelligence layer can perform centralized directory-based semantic searches [23] and find and install services that provide functionality to the application.

Such an architecture poses questions such as: how are the intelligence services hosted in the device? How is the intelligence functionality exposed to the applications (Cf. (b) in Figure 1.2)? How is the life cycle management of the intelligence services done? This thesis aims to explore what happens if an application requests (Cf. (a) in Figure 1.2) intelligence functionality that is not yet available in the device, a process that [38] calls Outward Intelligence Service Discovery (Cf. (c) in Figure 1.2).

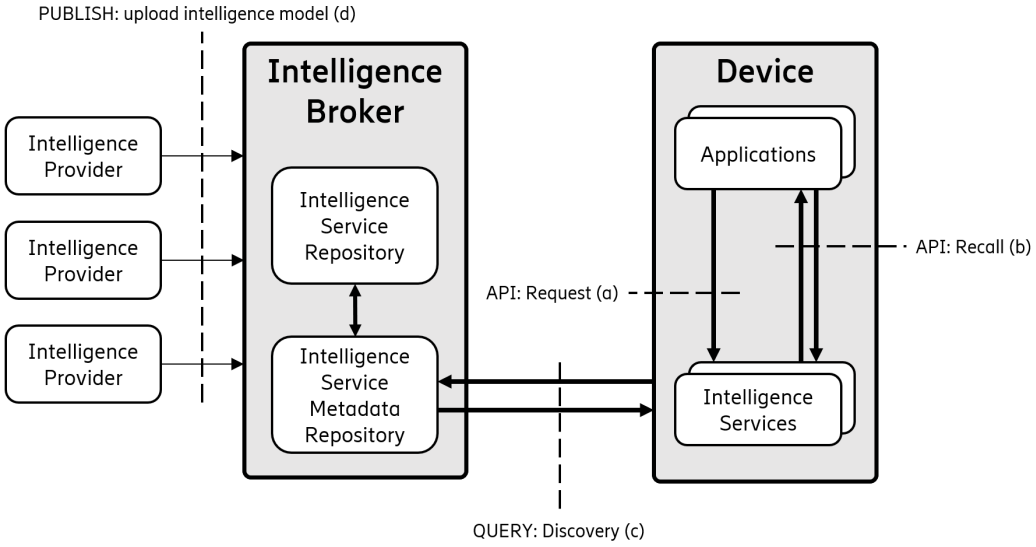

Figure 1.2: Overview of an ecosystem that enables the decoupling of intelligence functionality from applications in IoT devices.

As part of the Outward Intelligence Service Discovery, the intelligence layer contacts one or multiple intelligence service brokers to search for intelligence services that can fulfill the intelligence request from the application. This search does not only consider the requested intelligence service, also the device's capability, policies and specific domain constraints are important criteria and are part of the query. When a match is found with offered intelligence services, the results are sent to the intelligence layer of the device so that the user can pick one.

1.1 Problem Statement

The decoupling of intelligence functionality from applications has much to offer. It offers the potential of intelligence models to be searched for, installed, updated or retrained without the need for changes in the applications. These capabilities come in useful when an device owner or device manufacturer would like to update artificial intelligence functionality of the device. For example, a consumer buys a camera system with the ability to recognize when people enter their property through a machine learning algorithm. When the manufacturer improves the algorithm to make distinction between mailmen, children and other persons, the intelligence layer would take care of that upgrade.

This framework also opens the door value sharing. The manufacturer of the camera system might be known for its capable hardware but might not have the best intelligence algorithms. Specialized intelligence providers can monetize their intelligence algorithms through a centralized intelligence broker.

In practice any model could be installed in the intelligence layer, but without the right coordination, this functionality will only increase the amount of manual model selection that has to be done by the user to find the right models. Hence, the decoupling the intelligence models from the application will have to go hand in hand with compatibility measures. In the process of model discovery the offered intelligence models have to be matched with the search requests. Opposed to a web search request, there is some criteria that have to be accurately matched for the model to be able to do function as it is meant to. Therefore, in the process of intelligence model discovery, an uniform way of representing these criteria is necessary to be known with all stakeholders.

Here are some of the criteria that will be taken into account in this thesis. Machine learning models need the right execution environment to be available locally on the device. The execution of the models could potentially be done with the use of a runtime, a virtual machine, an interpreter or even compiled code. Though, there is many constrained IoT devices that will not have the capabilities to host these enablers. Information on the execution environment in the device needs to be modeled so that it can be taken into account in the query. There might be hardware requirements too. The right RAM size to load the data needs to be available and the CPU performance should be of such a level that the inference speed is adequate for the use of the model. Many machine learning models can only do a meaningful inference when the right data in the right structured format is fed to the model. The same goes

for the output data, the compatibility of the model and the application is strongly dependent on consensus of the structure and meaning of the data.

Lastly, and in the case of this thesis maybe most importantly, to let the user adequately search for models that fit his or her application there needs to be a formal description of what the models are trying to achieve. This is a complicated task to fulfil. There is many criteria, as listed above, that have to be taken into account. Many of these criteria can be captured with the common data types such as integers and strings. For example, conveying the information that a model takes input data in the form of a three-dimensional array with 28 by 28 float values is not very complicated. Describing the goal of an intelligence model will be harder to achieve as developers could virtually make models with any purpose.

1.2 Research Questions

The objective of this thesis is to research and implement the process of Outward Intelligence Service Discovery. The focus will be in finding a standardized way of representing meta-information about intelligence services. The implementation, that allows for storing and querying meta-data, serves as a test of the effectiveness of the solution and is not necessarily a goal on itself. The implementation stores meta-data of intelligence service models, according to the meta-data model, in a server that can be queried. This server represents an intelligence broker.

This thesis aims to reach this goal through answering the following research questions. There is a separation between questions that concern (a) the theoretical problem of representing the metadata of the intelligence services and (b) the engineering questions that concern the implementation.

a) How can Outward Intelligence Service Discovery be enabled through a metadata-model of Intelligence Services?

   - How to define intelligence services with a metadata-model?
   - What metadata schemas for services exist and do they suffice?
   - What are the shortcomings of these schemas in the context of this thesis?

b) How to implement the process of Outward Intelligence Service Discovery?

   - What metadata syntaxes allow for representation of intelligence services?
   - What query languages are fit for matching an intelligence request with intelligence services?
   - What database provides the functionality for storing and querying the meta data?

1.3 Structure of the Thesis

This thesis is about the outward intelligence discovery for IoT devices with an intelligence layer. Chapter 2 starts by giving some definitions often used in this thesis. Then it gives an explanation of ecosystem in which the intelligence layer operates and an overview of technologies that are used in the implementation. Chapter 3 breaks down the process of intelligence discovery and details how Intelligence Service Schema enables representation and discovery of intelligence services. Chapter 4 give an evaluation of the implementation, discusses its shortcomings and provides some suggestions for further research. Chapter 5 contains the conclusion of this thesis.

Chapter 2

Background

This chapter starts by explaining the main concepts behind artificial intelligence. Then an overview of the intelligence layer ecosystem is layed out. Lastly, an overview of the semantic web technologies used in the implementation is given.

2.1 Artificial Intelligence

The Cambridge English dictionary gives simple definition for intelligence:

> *The ability to learn, understand, and make judgments or have opinions that are based on reason.* [37]

Intelligence is a term that is quite broad and can be used in many different contexts. In this thesis, I will be using the word intelligence regularly in the context of artificial intelligence or more specifically machine intelligence. It is useful to take a closer look at the definition of these terms and how they are defined in leading literature.

The term artificial intelligence has been proven to be a loosely defined term. The artificial intelligence effect is called after the fluidity of the term [45]. The artificial intelligence effect takes place when a new algorithm solves a complex problem. In Machines Who Think, Pamela McCorduck says that "practical AI successes, computational programs that actually achieved intelligent behavior, were soon assimilated into whatever application domain they were found to be useful in, and became silent partners alongside other problem-solving approaches, which left AI researchers to deal only with the 'failures', the tough nuts that couldn't yet be cracked"[29]. This would define artificial intelligence as anything that has not been done yet.

Historically, the definitions of artificial intelligence were concerned with reasoning, behavior and reaching human performance with rationality as

ideal measure performance [41]. Haugeland and Bellman, respectively in 1985 and 1978, wrote that the goal was to make computers think and create "machines with minds", citing **thinking humanly** as goal [9, 21].

Slightly different definitions were offered by Charniak and McDermott, and Winston [12, 46], calling **thinking rationally** the goal of artificial intelligence. Winston said that artificial intelligence concerns "the study of the computations that make it possible to perceive, reason, and act" [46].

Rich and Knight, already understanding the artificial intelligence effect in 1991, wrote that artificial intelligence concludes "the study of how to make computers do things at which, at the moment, people are better", citing **acting humanly** as goal[40].

Finally, in an effort to narrow the definition, Poole et al. and Nilsson wrote that **acting rationally** is the goal of artificial intelligence [33, 36]. Poole et al. wrote that "intelligence is the study of the design of intelligent agents" and Nilsson wrote that artificial intelligence "is concerned with intelligent behavior in artifacts."

The idea that there would come an intelligent machine with the ability to act and think rationally or humanly was central in the field of AI. Though, the paradigm within the field of artificial intelligence has changed significantly, as Darwiche explains in [17]. The field of AI has been surging after a prolonged period of no breakthroughs. Machine learning techniques has lead the field into a new era that lead to students, even outside the fields of computer science and artificial intelligence, to practice and work with the techniques of machine learning and more particularly deep learning. These technique have proven to be extremely powerful for smaller constraint problems that can be mathematically defined. With this paradigm change, artificial intelligence researchers have been focussing more on solving smaller problems, opposed to the historical goals cited earlier, which brings us to machine intelligence.

2.1.1 Machine Intelligence

Machine intelligence does not have the goal to reach human level of intelligence [25]. Instead, machine intelligence is concerned with a smaller task that is considered intelligent. A algorithm using a live image stream to detect traffic signs would be considered machine intelligence. Such algorithms capitalise on the ability to mathematically formalize the problem at hand. Through intrinsic or acquired ability encapsulated in a computer program, the computer machine is able to carry out actions that achieve a predetermined goal. A task like the detection of traffic signs or translating text are specific enough to be considered machine intelligence tasks. An autonomous driving algorithm is not.

The ability to execute the task at hand might be represented by a function or model specifically fit for the task. The acquisition of this ability is done through a process called learning. In the case of machine learning, this is done through fitting a model to data. This learned model can then perform an intelligent task with novel data.

2.1.2 Machine Learning

Machine learning has lead to a breakthrough in artificial and machine intelligence. Expert systems, systems that emulate the decision making process of human experts, were often giving disappointing results. Expert systems were often expected and promised to solve complex problems as well or better than human experts even though this was rarely the case [8]. The rediscovery of the backpropagation algorithm, together with the increased availability of processing power of today, caused a resurgence in machine learning research resulting in algorithms outperforming human experts in some specific fields. Some example tasks where machine learning algorithms excel are language translation, activity detection or anomaly detection.

In machine learning many algorithms can be used to fit data to a mathematical model. Over time new data can be used as evidence to improve the model over-time. Machine learning problems covers three types of major problems:

- **Supervised classification**, concerns learning patterns in labeled sample observations in order to map new data points to categories (classes), e.g., predicting objects shown in an image.
- **Regression**, estimates the relationship (continuous value) of a dependent variable from one or multiple independent variables, e.g., the prediction of tomorrow's temperature.
- **Unsupervised methods (clustering)**, divides a set of data points into groups (clusters), which minimize the dissimilarity between samples in each group. Problems like pattern recognition and data structure discovery are based on clustering.

Furthermore, machine learning techniques can be divided in supervised or unsupervised. Supervised techniques require large amounts of labeled data to learn patters during the training process. Due to this problem specific training, supervised techniques can achieve good accuracy for that specific problem only and are barely reusable in different conditions. On the other side there are unsupervised techniques aimed to build models from unlabeled data.

2.2 Intelligence Stratum

The intelligence stratum architecture aims to move the machine learning functionality from the application layer and create an intelligence layer that runs a local service, say intelligence service. As the intelligence layer hosts the machine learning models, applications can make calls to the service. On the other hand, life cycle management is made much easier as the intelligence models are excluded from the applications. That is in a nutshell the value of the intelligence stratum. In this section, the architecture and features of the intelligence stratum will be explained as suggested by Ramos, Morabito, and Kainulainen [39].

The intelligence layer will interact with the following actors and provide services to them. Short descriptions of the functions of the several actors in the ecosystem of the intelligence stratum will provide a better overview of the architecture.

2.2.1 Intelligence Broker

The intelligence broker is a trusted rendezvous point and mediator between intelligence providers and the devices. The intelligence broker has the responsibility to manage the life cycle management of the distributed intelligence among the devices and supports the interactions between the actors within the ecosystem. It can be seen as an intelligence market. The broker provides the devices a single contact point through which the life cycle management of the intelligence services in the intelligence layer are is done. The devices register to the broker to gain a secure connection through which the device can make requests for discovery and provision of intelligence.

2.2.2 Applications

The applications are located in the application layer of the IoT devices. In the context of the intelligence stratum the applications are considered the users as they make calls on the intelligence services provided by the intelligence layer. What intelligence services are available in the intelligence layer depends on several aspects. Applications will communicate with the intelligence layer what services it needs. If these services are not yet available in the device, a process of intelligence discovery will be initiated in order to find out where such a service is available. The broker acts as the mediator through which intelligence services, provided by registered intelligence providers, can be navigated through, filtered and queried. When a suitable service is picked, the intelligence layer in the device will be provisioned with

the intelligence service. Intelligence services can be realized in the intelligence layer in different ways. One way is that a machine learning model is executed in a local runtime. A second option is provisioning the device with a piece of code is executed in a virtual execution environment that directly serves the application. A third option is a service that redirects every call from the application layer to a remotely executed intelligence function.

2.2.3 Intelligence Data Provider

When needed, Data Providers can provide data to the intelligence layer. For some models it is beneficial to retrain locally. When a device can not provide the intelligence service with all the data that is needed for a certain inference, it might be provided by an external Data Provider. When external data is needed, the intelligence layer must provide the application with storage, pre- and post-processing capabilities, and access control to ensure confidentiality and integrity. This Data Provider may be the same actor as the intelligence provider.

2.2.4 Intelligence Provider

The Intelligence Provider defines, configures, and initializes the intelligence service that will be made available to the application through the intelligence layer. It designs the algorithm. This could be a simple regression model that takes raw local data as input. On the other hand, an algorithm could consist of a pipeline that executes several pre-processing tasks before it executes a neural network classification task. The intelligence provider takes care of training of machine learning models if necessary or provides data for local training.

2.2.5 Intelligence Service

Within the intelligence layer, Intelligence Service can be seen as an Intelligence Actor. These services are composed by Intelligence Providers and later maintained through the life cycle management functionality. A service can be divided in two levels of granularity. A simple service might be a function that executes a normalization of a certain batch of data. This type of services is called Atomic Intelligent Services. The second type is called Fine-Grained Intelligent Services which are more complex services often existing out of a pipeline of multiple Atomic Intelligent Services. Intelligence Providers design these pipelines in a way that inputs and outputs are concatenated and

together form a more complete intelligence function. An example of a Fine-Grained Intelligence services is a pipeline of functions that together execute a linear regression including the needed pre- and post-processing of the data. The life cycle management applies to both the Atomic Intelligent Services as well as the Fine-Grained Intelligent Services. When updating intelligence actors, the dependencies created by the composition of services is kept track of by the intelligence layer. Through management policies updates and modifications of actors must be verified and authorized. Also, the executionâs environment of the Intelligence Actors have to be taken into account. Examples of devicesâ conditions that are significant for the execution of intelligence services are load, inference, concurrency, mobility state, internet connection, and memory. A capability management system keeps track of the capabilities of the device. The intelligence layer will then consider the requirements and performance metrics of intelligence services to deploy and find the best domain to deploy the Intelligence Actor.

2.2.6 Operations Control

Devices need configuration, monitoring and operation. All this happens through the Operations Control that is in direct control of the operations in the intelligence layer. Through the Operations Control policies for data handling, accessibility of the device' resources and functionality, and use of intelligence services are configured and maintained.

2.3 Semantic Web Technology

The intelligence stratum architecture relies on the high degree of interoperability between all the different agents that play a role int the ecosystem [7]. When the intelligence service is created by the intelligence provider, metadata about the intelligence service should be precisely registered and stored to be consulted for accurate intelligence service discovery. This ecosystem aims to be compatible with a large share of heterogeneous devices in the IoT domain. This means that the capabilities of the devices will vary and this has to be taken into account. Information about the capabilities and requirements of intelligence services will have to be matched with the capabilities and requirements of the devices to guarantee the compatibility.

Semantic Web technology can play a pivotal role in the intelligence stratum, an ecosystem of inter-connected agents. Semantic Web technology can be implemented and utilized in in the various agents to ensure interoperability. Researchers have suggested that, through the use of semantic data models, heterogeneous nature of devices and services in the IoT domain [6].

2.3.1 Resource Description Framework

The resource description framework (RDF) is a simple data model for processing metadata. By providing a framework for machine understandable descriptions of resources, RDF provides interoperabillity between applications. RDF can describe any resources that can be identified by a Uniform Resoure Identifier (URI) [34].

Statements, in the form of a triple, describe the resource. Properties are specified using predicates, pointing to other resources or literals, describing the resource. In Figure 2.1, *Document 1* (subject), *Author* (predicate) and *Author_001* (object) represent one triple. In other words, *Author_001* is the *Author* of *Document 1*. As shown in Figure 2.1, a set of multiple triples combined form a directed graph and allow the description of relevant information.

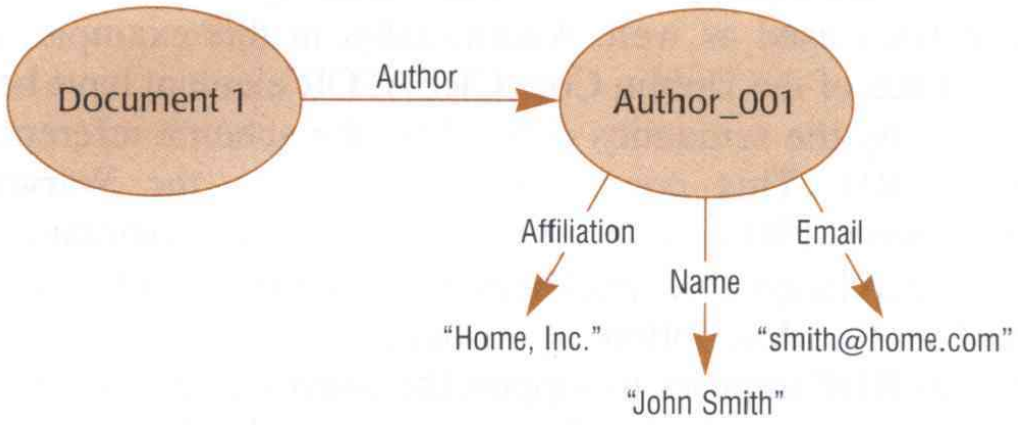

Figure 2.1: RDF example. Adapted from [30]

RDF allows communities to define semantics as a form of standardization. Metadata schemas allow communities to precisely define terminology for the description of resources of a certain domain. This is useful as terms might have a different meaning or implications when used in different contexts. For this reason, RDF allows for unique identification of the used semantics through the namespace mechanism. Instead of using the bare predicate "Author", users can look up a predefined vocabulary written by a governing authority. This improves interoperability and prevents ambiguity. For example, we could use terminology from Dublin Core (DC), a schema for representing metadata about web documents. Using the standardized predicate "creator", described by DC as "An entity primarily responsible for making the resource", instead of using own predicate for can improve interoperability [5].

RDF is a powerful yet simple model for the description of resources. Triples are written in a machine-readable files allowing communication among applications. Historically, RDF-XML is used as the syntax for RDF metadata. Nowadays, many other syntaxes are used, e.g., Turtle or JSON-LD. In this paper I will use Turtle, as it is the easily readable.

2.3.2 Semantic Web Services

Semantic web services are services offered through the web and advertised with a semantic description in machine-understandable form with schemas-based annotations. Through annotations, the web service description defines how to use the service and what it does [23].

Many specifications, languages and formats for semantic service descriptions exist. Prominent ones are OWL-S (Web Ontology Language for Web Services) [28], WSML (Web Service Modeling Language) [13], SAWSDL (Semantic Annotations for WSDL and XML Schema) [18] or Hydra [24]. These specifications serve slightly different purposes but all define web services. Therefore, these specifications are tightly coupled with their technology stack (HTTP-based, SOAP-based or defining RESTful APIs). In this thesis, we aim to present a data model and semantic schema that allows users to define intelligence services independently of its technology stack.

2.3.3 SPARQL: Semantic Query Language

SPARQL is a semantic query language that can be used to manipulate and query RDF data. It is standardized by W3C for information retrieval concerning semantic data. As RDF can be seen as graph data, SPARQL is essentially a graph matching query language. It supports multi-attribute and range queries of both natively stored RDF data and RDF via middleware (e.g. a triplestore).

SPARQL has grown to become the most popular technology concerning discovery services and is standardized as a W3C recommendation [20]. In "A Categorization of Discovery Technologies for the Internet of Things", Bröring et al. concluded that SPARQL is most fit for semantic discovery based directories due to its ability to produce rich queries and rank matched objects [11].

SPARQL uses four different query forms. All these query forms use pattern matching and return result sets or RDF graphs in several different formats, e.g., JSON-LD, Turtle or XML.

- **SELECT**, returns the variables requested or bound in a query pattern match.
- **CONSTRUCT**, returns an RDF graph with the requested variables in a form dictated by the query.
- **ASK**, returns *true* or *false*, indicating if the queried pattern exists or not.
- **DESCRIBE**, returns an RDF graph describing the queried resource.

Chapter 3

Implementation

In this chapter I will further break down the intelligence discovery in the intelligence layer. As this framework has many different agents, that might have different interests and policies, the process of intelligence discovery will be proven to be a complex one. First I will explain the distinction between intelligence discovery within the intelligence layer and outward intelligence discovery, show important aspects, such as policies, that has to be taken care of and how these processes could be implemented. In Section 3.1, I will describe the process of intelligence discovery within the intelligence layer. This is a process that encompasses interaction between a certain application and the intelligence layer. I will explain how a semantic API could be used to accommodate this communication. The basis for this semantic API is an intelligence layer data model according the the Resource Description Framework. In Section 3.2.5, a specification of this model will be given. Then I will describe an example implementation of the outward intelligence discovery. This implementation will serve as a proof of concept and demo of how the RDF intelligence service metadata model works in practice.

An important note to this chapter is that, at the point of writing this thesis, there is no implementation of the intelligence layer yet. My contribution to this framework will contain out of stating my vision of how the intelligence discovery, within the framework, as a whole would work best. Besides that, I deliver an implementation of the outward intelligence discovery.

3.1 Intelligence Discovery

Value sharing is one of the main features of the intelligence layer framework. Through intelligence brokers and providers, application developers can potentially be provided with greater supply of machine learning algorithms that they could use in their application without much effort.

Many machine learning algorithms are multi-purpose. A neural network can be used for many different goals. The other way around, for a certain goal, say facial expression recognition, many successful algorithms have been developed [42]. This is one of the challenges, although certainly not the only one, that makes autonomous search of goal-specific intelligence services complex. This thesis is an effort to simplify the discovery of goal-specific intelligence services.

An important step towards autonomous discovery of such services is having a data model for representing intelligence services. This data model needs to be able to contain all the necessary metadata about the intelligence service. This includes information about the requirements of the device, such as computing power, and metadata about the format of input and output data. But I also aim to provide a framework along which it would be possible to capture the goal of an intelligence service. Using this semantic model intelligence providers should be able to describe what their intelligence service aims to achieve. Once intelligence providers can describe the goal of their intelligence service, application developers can use that same intelligence service description framework to request intelligence services from the intelligence layer.

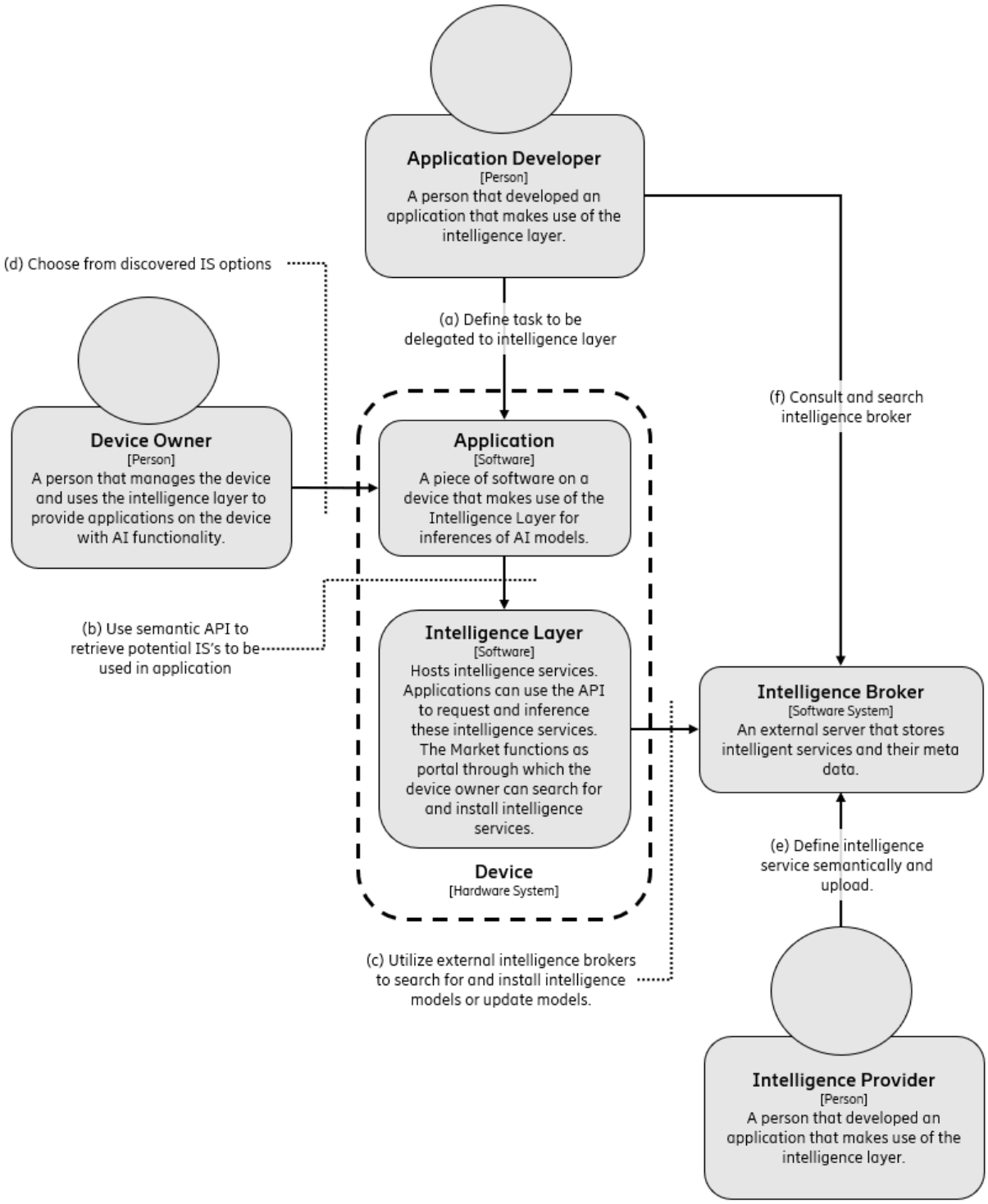

Figure 3.1: Context view of semantic API discovery process.

3.1.1 Discovery API

First of all, the semantic API provides functionality to search for intelligence services (Cf. (a) in Figure 3.1). Through this call, criteria and filters, such as input, output, goal and domain (see Section 3.1.4, Intelligence Service Selection by the Application Developer), can be set as parameters which are then sent to the intelligence layer.

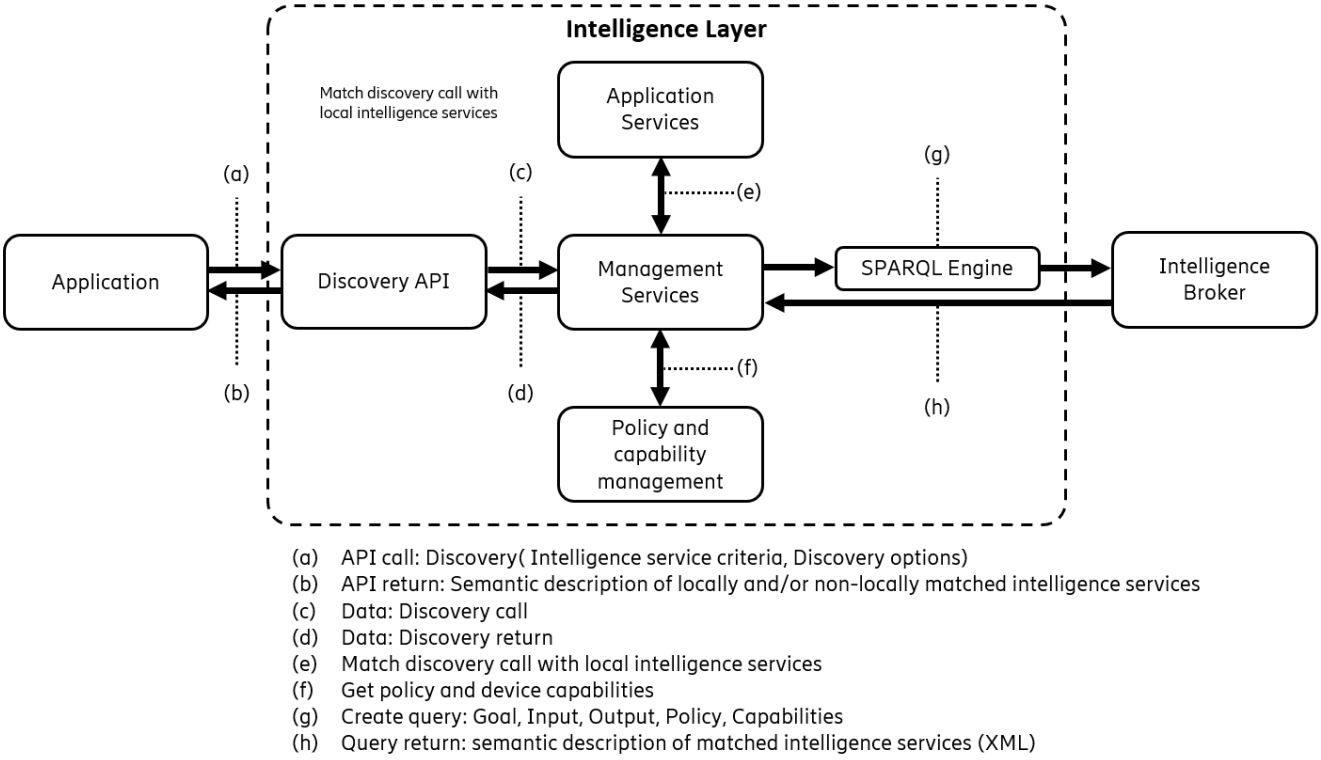

Figure 3.2: Process view of the intelligence service discovery.

It is possible that the intelligence layer already hosts intelligence services that match the criteria in the discovery call in which case outward discovery might not be necessary. This could be the case if application A uses intelligence service X and B is searching for similar functionality as X offers. In other cases, the user might want to do outward discovery only. Parameters in the discovery function will allow to set the preferred course of action.

When outward discovery is preferred, management services gathers information about the device's policy and capabilities (Cf. (e) and (f) in Figure 3.1.1. Policy and capability information, together with the application's dis-

covery criteria, form the criteria used to match with intelligence services' metadata in the intelligence broker.

The intelligence broker stores information of the available intelligence services in a metadata repository. Querying against this repository returns all necessary information about the intelligence service. The results of the query are returned to the application in RDF format (e.g. XML syntax).

The discovery process is not an obligatory process to take place in order to use the intelligence layer. It is possible that the developer already knows what intelligence service is fit for the application. The developer already has a default intelligence service that guarantees the correct workings of the application. In this case, the application is not required to use the discovery functionality, the application can simply request registration (Cf. (b) in Figure 3.1.3) for that particular intelligence service.

3.1.2 Registration API

With a registration call, the application requests access to recall functionality of a particular intelligence service. After the intelligence layer performed a policy and capability check the application gets registered for that particular. The intelligence layer creates a unique token for the registration, stores the application, intelligence service and token in a register and returns the token to the application. The token is utilized by the intelligence layer as an identifier of the application.

If the device's policies and capabilities allow it, it is possible for an application to register for multiple intelligence services. The application gets a new token for every registration.

Applications can unregister intelligence services. When an application no longer needs the intelligence service, the registration can be removed. In theory, the application is allowed to register and unregister to intelligence services as it sees fit. Unreasonable use of this functionality could be limited by device policies.

3.1.3 Recall API

With the received tokens the application can make intelligence recalls to the intelligence services hosted in the intelligence layer. An input, together with the right token, is sent to the application layer to do an inference for the application. The intelligence layer returns an the output of the calculation.

The input and output format are known to the application as the application has the intelligence service metadata. With this information the application knows the format and datatype(s) the input is supposed to have

in order for to make a correct inference as well as what format and datatype(s) the result of the inference will be.

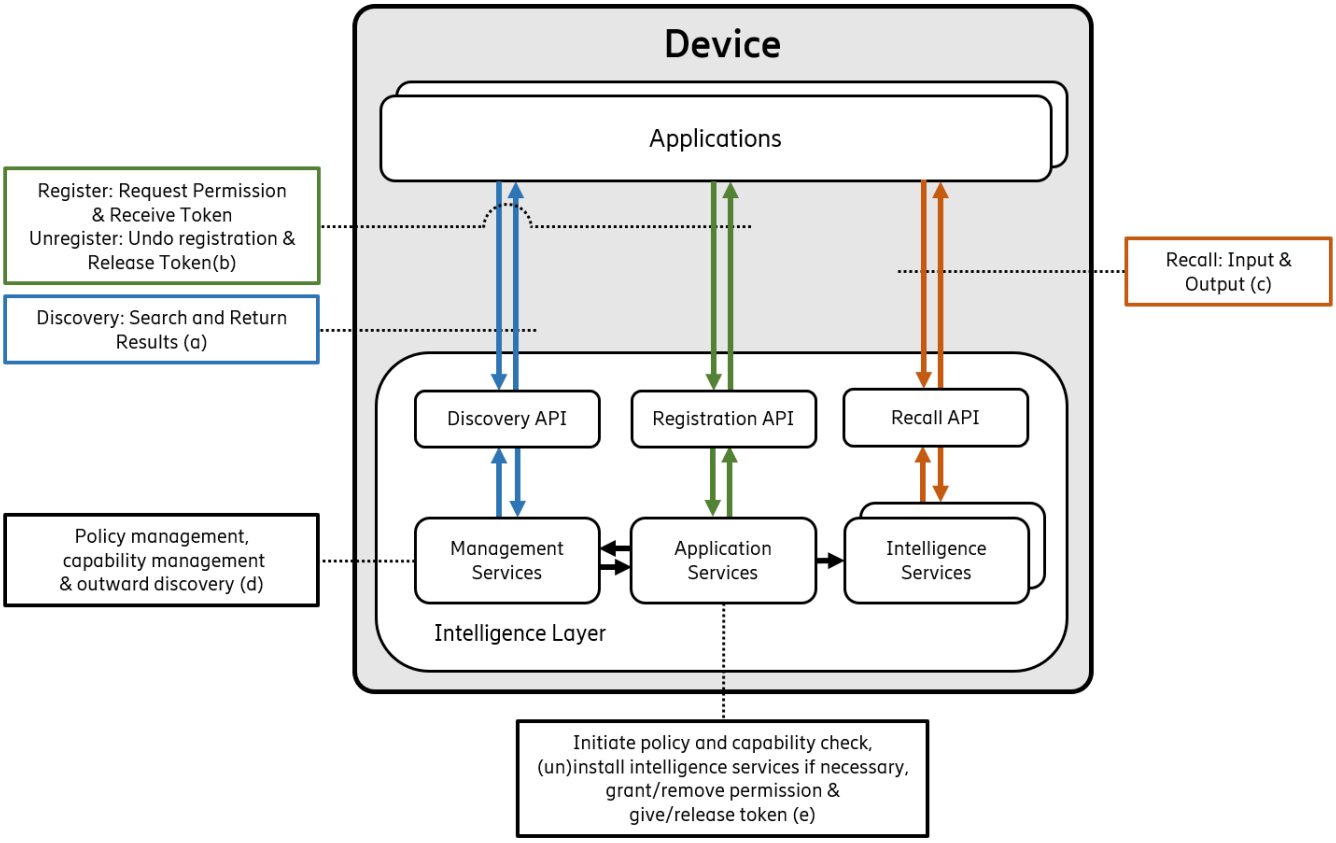

Figure 3.3: Context view of the intelligence service API functionality available to the application.

3.1.4 Intelligence Service Selection by the Application Developer.

When an application developers decides to use the intelligence layer for a certain task in their application, he or she should define that task using the intelligence service metadata model (Cf. (a) in Figure 3.1). The application developer is required to define three parts to be able to register to the intelligence layer. First, the problem needs to be defined. The problem definition is will be a combination of predefined and publicly available semantic concepts (later in this thesis I will explain this further together with some examples).

Secondly, to make the installments and execution of the intelligence services seamless, the inputs and outputs of the intelligence services needs to fit the application very precisely. Therefore, precise information about the needed format of input and output data need to be included in the request.

With this task defined, the application can make a discovery call and get a list of the matched intelligence services. How this list of results from the discovery call is handled and how the selection if the intelligence service is made, is for the application developer to decide. Here is some options with considerations.

An autonomous choice could be made with predefined criteria hard-coded in the application. An application developer could only be interested in intelligence services from a certain intelligence provider. This intelligence provider, say a certain company with a good reputation concerning the intelligence layer, might have a number of intelligence providers for different tasks. The task definition together with the name of the intelligence provider could be enough to autonomously find the right intelligence services.

Another option is to involve the device owner in the process of intelligence discovery. The functionality of intelligence discovery through the intelligence layer could be seen as seen as an interesting feature to the application user that adds increased flexibility in the app's usage. Through a custom user interface the application developer could allow the user to set some of the discovery criteria and pick their preferred option. There might be multiple intelligence services that achieve the same task but with different performance.

3.2 Intelligence Service Metadata Model

The intelligence service metadata model allows intelligence providers to declare that a certain intelligence service exists and store critical information for execution of the intelligence service. It describes the mapping from input data to output data, but without describing the internals of the functions. It describes execution related requirements for the intelligence service to run on the device. Furthermore, a task definition of the intelligence service allows the intelligence developer to semantically describe the goal of the service.

The intelligence service metadata model consists of several classes. Instances of these classes together describe one intelligence service. It is possible that one intelligence includes multiple instances of one particular class, e.g. an intelligence service can consist of multiple atomic services.

In this section, I will lay out the classes of the metadata model and describe the purpose of them. A full RDF Turtle serialization of the intelligence service metadata model can be found in Appendix B.1. In Subsections 3.2.1 and 3.2.2, I will explain the classes forming the skeleton of this data model by using of parts of the DeepSpeech example in Subsection 3.2.5.

3.2.1 Intelligence Service

This class declares an intelligence service and its metadata. An intelligence service is a service, hosted by the intelligence layer. Through the api of the intelligence layer an intelligence service takes input from an application. It then executes a set of calculations and returns an output to the application. This class inherits properties datatype properties from its parent class `uniq ue-entity`. These datatype store information about the intelligence service and can be queried in the discovery process.

For the demonstration of the Intelligence Service Schema an intelligence service called DeepSpeech Plain English will be used. DeepSpeech Plain English is an intelligence service that offers speech recognition functionality. More specifically, the service is specialized in recognizing English commands for personal assistants, e.g. checking the calendar, creating appointments or calling somebody. Subsection 3.2.5 goes into further detail about this example.

In Listing 3.1 it is shown how the intelligence service is declared using the Intelligence Service Schema. DeepSpeech Plain English is a `owl:NamedInd ividual` of the `iss:intelligence-service` class. The intelligence service class allows a unique identifier, name, description, version, date of creation, goal and domain to be declared. Furthermore, the predicates `iss:uploade`

dBy and hasImplementation declare a relation with named individuals of respectively class iss:account and iss:implementation.

```
:deepspeech_plain_english a owl:NamedIndividual, iss:intelligence-service ;
  iss:identifier "3384"^^xsd:string ;
  iss:hasName "DeepSpeech"^^xsd:string ;
  iss:description "DeepSpeech for personal secretary services in English"^^xsd:string ;
  iss:hasVersion "1.3.2"^^xsd:string ;
  iss:created "01-01-2020"^^xsd:date ;
  iss:hasGoal iss:category-prediction ;
  iss:hasDomain iot:speechRecognition ;
  iss:uploadedBy [ a owl:NamedIndividual, iss:account ; ] ;
  iss:hasImplementation [ a owl:NamedIndividual, iss:implementation ; ] .
```

Listing 3.1: Intelligence Service: DeepSpeech Plain English

3.2.2 Implementation

An implementation contains information about how an intelligence service is implemented. The underlying idea behind the separation of the implementation class and the intelligence service class is that an intelligence service could potentially have multiple implementations. In the DeepSpeech example, the only implementation defined uses a neural network to do speech recognition. The property iss:hasMethod

An intelligence service with a particular goal might have implementations for devices with different capabilities. Properties in the implementation class, e.g. hasRequiredRam and hasRecommendedCpuScore, can be used by the intelligence provider to define hardware requirements for a certain implementation.

Machine learning algorithms can differ significantly in their required resources. Different implementations will have a different hasServicePipeline. An hasServicePipeline is a rdf:List [10] which functions as an ordered array of URI's, in this case atomic-service's.

As intelligence providers create and publish intelligence services, it will be possible to use external intelligence services as part of another intelligence service. The property usesExternalService can be used to declare the usage of an external service as part of an implementation. The intelligence layer then knows that the installation of that external sevice is required.

Listing 3.2 demonstrates how an implementation of the DeepSpeech Plain English intelligence service is declared.

```
:DeepSpeechImplementation a owl:NamedIndividual, iss:implementation ;
  iss:hasIdentifier "9283"^^xsd:string ;
  iss:hasName "DeepSpeech Personal Secretary"^^xsd:string ;
  iss:hasVersion "1.3.2"^^xsd:string ;
  iss:hasDescription "This implementation uses a deep neural network."^^xsd:string ;
  iss:created "01-01-2020"^^xsd:date ;
  iss:hasImplementationScore "89"^^xsd:integer ;
  iss:hasMethod iss:neural-network ;
  iss:requires  [ a owl:NamedIndividual, iss:dependency ; ] ;
  iss:requires [ a owl:NamedIndividual, iss:dependency ; ] ;
```

```
11  iss:hasServicePipeline  ( [ a owl:NamedIndividual, iss:atomic-service ; ] ) .
```

Listing 3.2: DeepSpeech Plain English implementation: DeepSpeechImplementation

There is several approaches to describing a goal of an intelligence service. From an high-level point of view we could say that a service is merely a mapping of input data to output data. This is true for every function-based service and can be done by describing the input's and output's data format. The Web Service Description Language did this successfully for web services [13]. With this information applications know how to execute a service.

From the data structure and format we cannot always derive the goal of the service. From an machine intelligence point of view, we could give extra meaning to intelligence services by capturing what task the main algorithm tries to solve. In a recent paper called "Machine learning for internet of things data analysys: a survey", Mahdavinejad et al. [27] discussed how a taxonomy could help find the right algorithm for a problem. Mahdavinejad et al. found the following categories to be useful for describing the task of machine learning algorithms in the Iot domain:

- To discover the structure of unlabeled data;
- to find unusual data points and anomalies;
- to predict values and classified sequenced data;
- to predict the categories of data; and
- to extract features of data.

These categories state the goal of machine learning algorithms, but indirectly also the mapping of the service. The intelligence service schema uses these categories as semantic concepts to describe the services' goal.

Lastly, this schema also aims to describe intelligence services on a lower level and allows users to describe their intelligence services with semantic descriptors from taxonomies of relevant domains. These semantic annotations will give the machine readable mapping, the data structure and format of the mapping, a human understandable context. In other words, not only the data format of the input and output are described, but also what the input and output data represents and in what context. I will give specific example, which I will go into detail about in subsection 3.2.5. DeepSpeech is a speech recognition service that takes audio as an input and returns the recognized speech in textual form. In the example the input and output are annotated with semantic descriptors and concepts from the Iot Domain

and the Speech and Language Domain. These domains are represented by their own taxanomies. These taxanomies is a set of concepts organized in a hierarchy starting with the most general concept and gets more specific and defined as you go down the hierarchy. Figure 3.4 and Figure 3.5 are simplified visualizations the respectively the Iot Domain and the Speech and Language Domain. These annotations can be used as filters in the SPARQL query and allow for precise and effective intelligence service discovery.

3.2.3 Domain Specificity

The goal of the intelligence service schema is to enable rich and efficient description of intelligence services from any domain or context. A rich description would mean that any intelligence service can be described in a precise manner. Efficient description would exist out of descriptors from standardized taxonomies that can be effectively used in the discovery process.

To allow for rich description of intelligence services, a wide variety of semantic descriptors should be available to the intelligence provider to describe their services. Chun et al. [14] argued that semantic information models aimed to describe heterogeneous resources, should be modeled with the use of ontologies. These ontologies can represent a set of common concepts relevant to the context of the resource that is being described. Such domain specific ontologies can provide formal expressiveness, avoid ambiguity in the description and increase the ability to analyze and inference the resources autonomously.

Domain specific taxonomies with context specific descriptors will also allow for more efficient description of the intelligence services. For the discovery process to be efficient the intelligence providers is required annotate their intelligence services in such a way that the intelligence services can be matched with the requests from the applications. With a standardized number of taxonomies, each specifically developed for effective semantic description of intelligence services from a certain context, ambiguity can be minimized [44]. Therefore, domain specific description can increase the interoperability among service providers, service consumers, data providers, data consumers and facilitate semantic interpretation and service integration.

Existing ontologies are often not specific enough to fully rely upon. Therefore, multiple taxonomies may be necessary to describe an intelligence service adequately and those taxonomies might have overlap. In the case of the DeepSpeech example, the Iot taxonomy and the Speech and Language taxonomy, respectively Figure 3.4 and Figure 3.5, were used to describe the intelligence services. These are simplified examples of such domain taxonomies, but show how multiple domain taxonomies together provide sufficient semantic

descriptors for rich annotation.

These taxonomies should, as much as possible, be put together with semantic descriptors from existing ontologies. Overlap between ontologies can be overcome with the use of built-in OWL properties such as `owl:sameAs`, `owl:equivalentClass` and `owl:equivalentProperty` [31]. Within domain taxonomies, subdomains could be modeled to further increase specificity. Such hierarchical modeling allows reasoning to take place at the intelligence broker's server side at after it received the query request from a device.

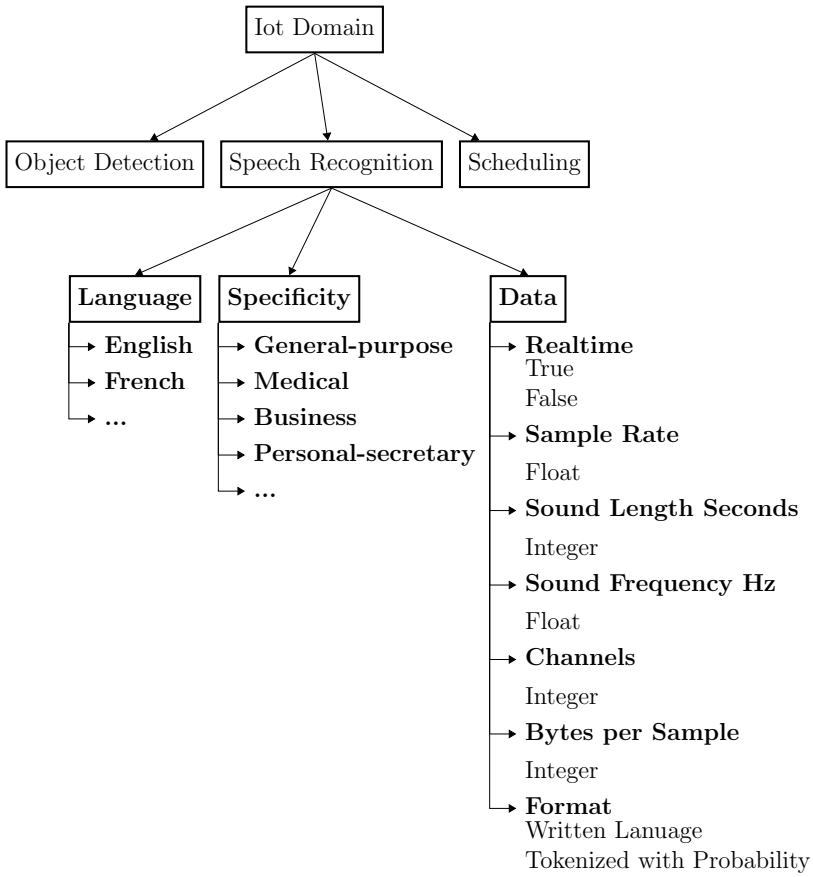

Figure 3.4: An example domain taxonomy that can be used to annotate intelligence services. Using a hierarchical structure the Iot domain has several subdomains; Object Detection, Speech recognition and Scheduling. Those subdomains contain descriptors that can be used to annotate intelligence services. In some cases values are categorical and are modeled as concepts (E.g. Language and Specificity). In other cases the descriptors come as datatype properties.

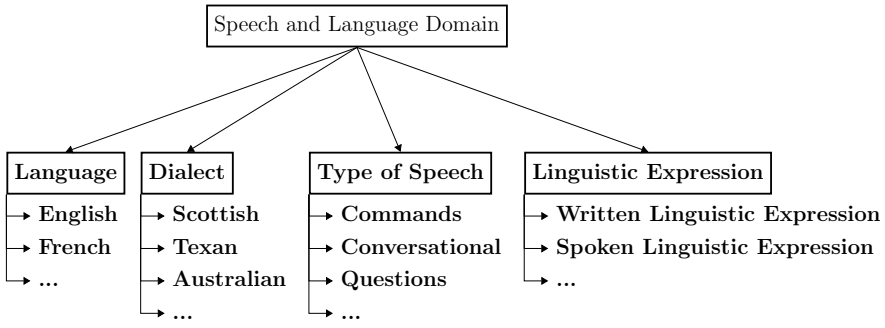

Figure 3.5: A simplified example domain taxonomy intelligence providers could use to annotate their intelligence services with semantic descriptors from a speech and language context.

3.2.4 Classes

These are the classes of the intelligence service schema. Figure 3.6 visualizes the relationships between the classes.

- `unique-entity`: Unique entities, such as intelligence services, atomic services, dependencies, execution environments, and implementations, inherit datatype properties from this class.
- `intelligence-service`: This class declares an intelligence service and its metadata. An intelligence service is a service, hosted by the intelligence layer, that takes input from an application, executes a set of calculations and returns an output in the context of the intelligence layer.
- `account`: Intelligence services can be uploaded through and managed by an online account. The account can be owned by, and therefore represent, a person or company.
- `person`: This class represents a person. In this context a person can be the creator of an intelligence service or holder of an account.
- `goal`: The goal of the intelligence service from a machine learning perspective.
- `method`: The method implemented to achieve the goal of the intelligence service.
- `domain`: The domain provides context in which the intelligence service operates. A domain taxonomy contains semantic descriptors used for input and output annotation.

- `implementation`: An intelligence service can have multiple implementations that serve the same goal. Different implementations could implement different atomic services, environment requirements, and data input and outputs.
- `execution-environment`: The execution environment contains information about the system that executes the code.
- `dependency`: Dependencies describe software modules necessary to execute the implementation.
- `atomic-service`: A service that is directly invokable. An atomic service has no subprocesses and is executed in one step. A pipeline of atomic services can form an implementation of an intelligence service.
- `dataset`: Contains information about the dataset that is used to train the algorithm.
- `label`: Labels of categories captured by the dataset.
- `data-entity`: Contains metadata about a data object. A data entity can be described by a domain and the semantic descriptors in the domain's taxonomy.
- `input`: Describes the input the the intelligence service requires from the application. An input can be described by a domain and the semantic descriptors in the domain's taxonomy.
- `output`: Describe the output the intelligence service returns to the application. An output can be described by a domain and the semantic descriptors in the domain's taxonomy.
- `annotated-entity`: A dataset, goal, output, and input are subclasses of annotated entity. These classes inherit properties from annotated entity.
- `data-tensor`: This class represents a data tensor of an input or output. A tensor has at least one dimension.
- `dimension`: A data tensor can have has at least one dimension. Every dimension stores an amount of values.
- `data-category`: The category of data. Either audio, text, image or tensor. Respectively, the first three classes are subclasses of tensor.

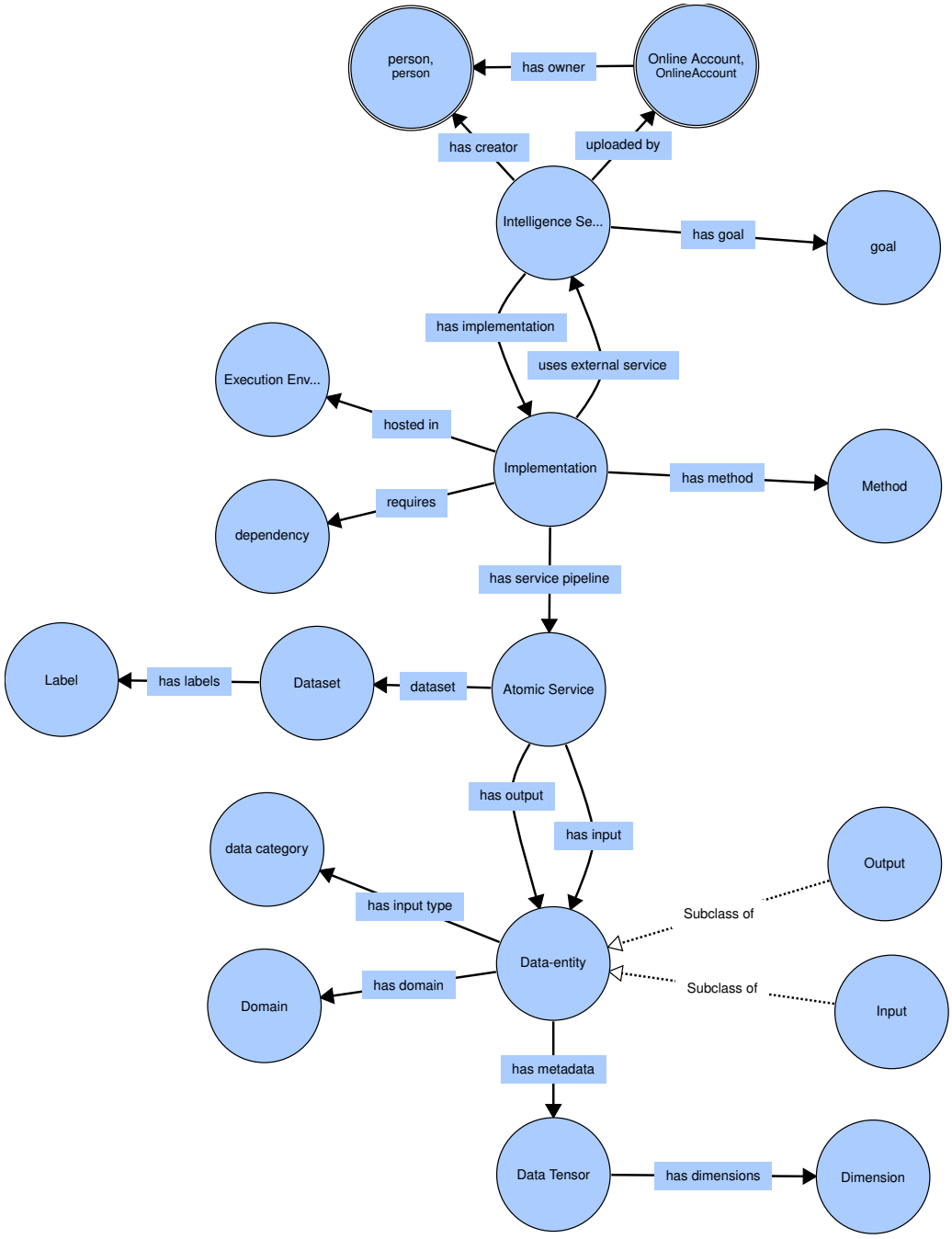

Figure 3.6: Visualization of classes of the the Intelligece Service Schema and their relationships. Tool for visualization: WebVOWL 1.1.7 [26]

3.2.5 Example: DeepSpeech Plain English

This subsection lays out how the Intelligence Serevice Schema can be used to semantically describe an intelligence service using RDF. The example concerns the DeepSpeech Plain English intelligence service. This description, fully listed in Appendix A, contains the metadata about the intelligence service that is used for the discovery of the service. After discovery and successful installation of the service in the intelligence layer, this service can be called by applications through the semantic api. In short, this service retreives the audio file, filters out the noise of the audio, carries out speech recognition and return the result in textual form through the api to the application. `:deepspeech_plain_english` is a definition of an intelligence service.

The intelligence service `:deepspeech_plain_english` has as goal `iss:categoryprediction` and is uploaded by `:account7466`. This account is owned by `:person8293`, whose name is `"HenkJanSmits"` and has a reputation score of `"77"`. These values are attached to the account and can be used as a filter in the search for services.

```
:deepspeech_plain_english a owl:NamedIndividual, iss:intelligence-service ;
                          iss:identifier "3384"^^xsd:string ;
                          iss:hasName "DeepSpeech"^^xsd:string ;
                          iss:description "DeepSpeech for personal secretary services in plain English"^^xsd:string ;
                          iss:hasVersion "1.3.2"^^xsd:string ;
                          iss:created "01-01-2020"^^xsd:date ;
                          iss:hasGoal iss:category-prediction ;
                          iss:hasDomain iot:speechRecognition ;
                          iss:uploadedBy :account7466 ;
                          iss:hasImplementation :NeuralNetworkImplementation .
```

Listing 3.3: This listing shows how the DeepSpeech Plain English service is described.

One implementation of the service is defined as `:NeuralNetworkImplementation` on line 1 in Listing 3.4. The implementation `:NeuralNetworkImplementation` has the name `"DeepSpeechPersonalSecretary"`. The method of this implementation is `iss:neural-network`.This implementation executes a pipeline of two atomic services, `:wavcleaner` and `DeepSpeechOnnx`, as declared on line 11 in Listing 3.4.

```
1  :NeuralNetworkImplementation a owl:NamedIndividual, iss:implementation ;
2                               iss:hasIdentifier "9283"^^xsd:string ;
3                               iss:hasName "DeepSpeech Personal Secretary"^^xsd:string ;
4                               iss:hasVersion "1.3.2"^^xsd:string ;
5                               iss:hasDescription "This implementation uses a deep neural network."^^xsd:string ;
6                               iss:created "01-01-2020"^^xsd:date ;
7                               iss:hasImplementationScore "89"^^xsd:integer ;
8                               iss:hasMethod iss:neural-network ;
9                               iss:requires :PythonDependency ;
10                              iss:requires :OnnxruntimeDepencency ;
11                              iss:hasServicePipeline ( :wavcleaner :DeepSpeechOnnx ) .
```

Listing 3.4: Definition of the service's implementation.

`:wavcleaner` is the first atomic service in the service pipeline of this implementation. Therefore, `lst:wavcleaner` contains metadata about the input of the whole service. The second and last atomic service, `:DeepSpeechOnnx`, contains metadata about the output. Listing 3.5 shows how an atomic service is described and how the input is attached to the atomic service (line 5 of 3.5).

```
1 :wavcleaner a owl:NamedIndividual, iss:atomic-service ;
2             iss:hasIdentifier "2783"^^xsd:string ;
3             iss:hasName "Wav cleaner"^^xsd:string ;
4             iss:hasDescription "This atomic service takes the location of an wav file as
      input and reduces noise in the audio."^^xsd:string ;
5             iss:hasInput :WavInput .
```

Listing 3.5: This listing shows how the atomic service `:wavcleaner` is declared.

The implementation requires the dependencies `PythonDependency` and `OnnxruntimeDepencency`, respectively declared on line 9 and 10 of Listing 3.4, to be available on the device. The description of the `PythonDependency` is shown in Listing 3.6. It stores the name and version necessary of the dependency necessary for the service to be functioning.

```
1 :PythonDependency a owl:NamedIndividual, iss:dependency ;
2                   iss:hasName "Python"^^xsd:string ;
3                   iss:hasVersion "3.7.1"^^xsd:string .
```

Listing 3.6: This listing shows how the Python dependency is declared.

The description of the format is required for autonomous communication between applications and the api. The `data-tensor` stores metadata about the in- and output of a service. Listing 3.7 shows how `WavInput`'s data format is described by a `data-tensor` (line 3). This tensor stores information about the format of the input. A tensor could have multiple dimensions but since `:wavcleaner` takes a file as input, the tensor only has one dimension (line 5) with one value (line 8) storing the path to the audio file. This is declared as follows:

```
1 :WavInput iss:hasMetaData :locatorDataTensor .
2
3 :locatorDataTensor a owl:NamedIndividual, iss:data-tensor ;
4                    iss:elementDataType xsd:string ;
5                    iss:hasDimensions ( :dimension0 ) .
6
7 :dimension0 a owl:NamedIndividual, iss:dimension ;
8             iss:dimensionValue "1"^^xsd:integer .
```

Listing 3.7: This listing shows how the input's format is described using the data-tensor class.

In order to describe the service's expected input, the input is annotated with semantic descriptors the Intelligence Service Schema and the two example domain taxonomies, the `iot:speechRecognition` domain and the `lin:linguistics-domain` domain, declared respectively on line 6 and 13 in the

Listing 3.8. These annotations describe the functioning of the service and can be used as filter in the discovery query.

The input has a `iss:hasDataCategory` of `iss:audio`, on line 4 in Listing 3.8, declaring that the service expects audio data in some form. The descriptors are used to state that this service is specifically effective recognizing spoken questions and commands (respectively line 16 and 19) typically spoken to personal secretary devices (line 9). The algorithm is specifically trained to process Australian English (line 17–18) language. Furthermore, the service takes the audio input in the form of a WAV-file with a sample rate of 48000Hz (line 10-11).

```
1  :WavInput a owl:NamedIndividual, iss:input ;
2            iss:isFile "true"^^xsd:boolean ;
3            iss:hasMetaData :locatorDataTensor ;
4            iss:hasDataCategory iss:audio ;
5
6            iss:hasDomain iot:speechRecognition ;
7  # The following properties are part of the
8  # Iot domain, subdomain: speechRecognition.
9            iot:specificity "personal-secretary"^^xsd:string ;
10           ebu:sampleRate "48000"^^xsd:integer ;
11           ebu:hasAudioFormat "WAV"^^xsd:string ;
12
13           iss:hasDomain lin:linguistics-domain ;
14 # The following properties are part of the
15 # Linguistics domain.
16           gold:LinguisticExpression gold:SpokenLinguisticExpression ;
17           gold:Language "English"^^xsd:string ;
18           gold:Dialect "Australian"^^xsd:string ;
19           lin:typeOfSpeech lin:Command, lin:Question .
```

Listing 3.8: Definition of the service's input.

3.3 Implementation: Metadata Repository

For the implementation of the outward intelligence discovery I have chosen to work with Apache Jena Fuseki [2]. Jena is an open-source framework that allows for the manipulation of RDF graphs [3]. Fuseki adds an HTTP interface to the server which allows remote querying and updating through ARQ, the SPARQL engine of Jena [1].

Jena Fuseki comes with TDB, a data base that stores semantic triples [4]. A TDB dataset contains one or more graphs, which contain triples. In the implementation of the intelligence service schema, every intelligence service is a named graph. A SPARQL query can be matched against all intelligence service graphs and return the matches.

When the intelligence layer receives the request from an application, the SPARQL engine creates a query for discovery (Cf. (g) in Figure 3.1.1). The SPARQL engine gathers the relevant data about the device, such as capabilities and policies, and autonomously creates a query. As shown in Listing 3.9, a SPARQL query exist out of triples containing information to be matched with intelligence service descriptions hosted by the intelligence broker. Because of the structural simplicity of SPARQL queries, it is simple to programmatically put a query together based on the data offered to the SPARQL engine. Variables can be left out of the query in case they are considered irrelevant. Starting from a baseline query, lines with necessary namespaces and to be matched triples and filters can be appended to the query.

Listing 3.9 is showing just one example of how a query, sent from the intelligence layer, could look like. Essentially, this query will match graphs and return the variables used in the query from those graphs. The strictly relavant variables are the identifiers of the intelligence service and the implementation of the matched intelligence services as those will be necessary for the installation process. Another option is to use returned variables to autonomously choose the best intelligence service.

This query will return all variables used in the query. SPARQL is an elaborate query language that allows for the use of a vast variety of matching patterns. As shown in Listing 3.9 line 7–8, this query only matches services newer than 2019-01-01. Besides certain required variables for a match, this query only match intelligence services that have a recommended RAM memory lower than 150Mb. The intelligence service is required to work well with Questions as a type of speech. With the OPTIONAL match pattern, on line 24, the query will return any other type of speeches the intelligence service works well with.

```
SELECT * WHERE
{
  GRAPH ?g
  {
    ?service iss:hasName ?name .
    ?service iss:identifier ?intelligenceserviceid .
    ?service iss:created ?date .
    filter ( ?date > "2019-01-01"^^xsd:date ) .
    ?service iss:hasDomain iot:speechRecognition .
    ?service a iss:intelligence-service .
    ?service iss:hasGoal iss:category-prediction .
    ?service iss:hasImplementation ?implementation .
    ?implementation iss:hasVersion ?implementationid .
    ?implementation iss:hasRecommendedRam ?ram .
    ?implementation iss:hasRecommendedCpuScore ?cpuscore .
    FILTER ( ?ram < 150000000 ) .
    FILTER ( ?cpuscore < 90 ) .
    ?input a iss:input .
    ?input iss:hasDataCategory iss:audio .
    ?input iss:isFile "true"^^xsd:boolean .
    ?input iot:specificity "personal-secretary"^^xsd:string .
    ?input gold:Language "English"^^xsd:string .
    ?input lin:typeOfSpeech lin:Question .
    OPTIONAL { ?input lin:typeOfSpeech lin:Command } .
    ?output a iss:output .
    ?output iss:hasDataCategory iss:text .
    ?output gold:LinguisticExpression gold:WrittenLinguisticExpression .
  }
}
```

Listing 3.9: Example intelligence service discovery query

Chapter 4

Evaluation and Discussion

The implementation as presented in this thesis solves the problem initially described for the DeepSpeech example. However, this thesis just contains one use case of the Intelligence Service Schema presenting the functioning of the schema. In practice, this example might prove itself to be more useful as an example of how to implement this first version of the Intelligence Service Schema rather than a proof that it will function in production. Further testing will have to be done before a full evaluation of the solution can be made.

The Intelligence Service Schema was developed as a part of an ambitious project in Ericsson's internet of things research and development department that aims to decouple intelligence from applications in devices. Thusfar, this project has been in conceptual state and Intelligence Service Schema is a first step towards a semantic schema that can be used to describe intelligence services in the multi-agent environment of the internet of things.

The Intelligence Service Schema allows intelligence services to be semantically annotated according to their goal independent of the technology stack. Well known languages and formats, such as OWL-S, WSML, WSDL and SAWSDL, serve a similar purposes although they all are tightly coupled with their web technology stack. The Intelligence Service Schema is technology independent. Another important feature of the schema presented in this thesis is the capability to annotate services with external taxonomies. Only SAWSDL shares this capability [18].

The outward discovery of intelligence services is detailed, though many other topics, e.g., the installation and execution of the intelligence services, have to be addressed and implemented to successfully allow decoupling of intelligence in internet of things applications. Besides that, more use cases should be created to further test the Intelligence Service Schema and consider additional improvements accordingly.

The decoupling of the intelligence layer comes with complications of which many have to be further researched. These are challenges relevant for future research relevant to the intelligence discovery process:

- **Policy Check in Intelligence Discovery:** policies of devices, applications and intelligence services will have to be managed within the intelligence layer. The description of these policies are not covered by the Intelligence Service Schema but still will have represented in a common language and coordinated.

- **Scalability of the Intelligence Broker:** Klusch et al. writes that where a centralized semantic search directory can serve as a intermediary between service providers and consumers, it also represents a potential single point-of-failure [23]. A *decentralized directory search* for semantic search, e.g. a structured peer-to-peer network with a query routing protocol, or *directory-less search*, e.g. an unstructured peer-to-peer network without an overlay structure, are potential alternatives that should be investigated. Such distributed solutions might be more suitable for real-world scenarios for their high scalability and low dependability [15, 22, 35].

- **String Search:** String search functionality can improve the results of outward discovery queries or explore intelligence services. String search would be useful, e.g., for browsing intelligence services through partial matching of intelligence service titles and descriptions.

- **Partial Matching:** In the case of no match, partial matching can improve the further search process. Systematic prioritization of matched triples needs to be implemented in order to rank the partial matches.

Chapter 5

Conclusions

The goal of the thesis was to find a technology stack independent semantic schema that can be used to define intelligence services and therefore allow discovery of intelligence services. Existing schemas were not exactly fit for this purpose. The Intelligence Service Schema, presented in this thesis, should be able to describe all information necessary to find out if intelligence services would be fit for an application on a given device. This is done through providing a set of semantic classes and properties to describe dependencies, runtimes, intelligence services, implementations, accounts, inputs, outputs, files and metadata of all these classes.

Besides that, this schema also aims to provide a framework of systematic semantic annotation in order to describe the intelligence service's goal. Through semantic annotation of the input and output, the semantic definition of the mapping of the service's function, the goal is described. The semantic descriptors can be taken from the domain specific taxonomies that will have to be defined in the future. In my thesis you will find an example taxonomies and the implementation. When intelligence service's are defined, they're stored and queried from a Jena Fuseki server, a semantic repository, using SPARQL.

The intelligence layer is a promising project and the Intelligence Service Schema is a step towards autonomous discovery of intelligence services. However, it is an innovation that is mostly in conceptual stage and it needs further work. One of the challenges would be to define domain specific taxonomies that can describe inputs and outputs. It might be fairly easy to come up with a taxonomy that could describe some example intelligence services. Though, in the past, promises of semantic technology have often times been bigger than the results, hence the need for development and testing of the Intelligence Service Schema. It is considered to be difficult, even for field experts, and time consuming to make useful domain ontologies (specific enough to

be useful, not too specific so that it's too difficult and unworkable). Defining such taxonomies with experts would be key to progress in the field of semantic intelligence discovery and description.

Bibliography

[1] ARQ - A SPARQL Processor for Jena. URL https://jena.apache.org/documentation/query/.

[2] Apache Jena Fuseki. URL https://jena.apache.org/documentation/fuseki2/index.html.

[3] Apache Jena. URL https://jena.apache.org/.

[4] Apache Jena TDB. URL https://jena.apache.org/documentation/tdb/.

[5] DCMI Metadata Terms, Jan 2020. URL https://www.dublincore.org/specifications/dublin-core/dcmi-terms/.

[6] M. Aziez, S. Benharzallah, and H. Bennoui. Service discovery for the internet of things: Comparison study of the approaches. In *2017 4th International Conference on Control, Decision and Information Technologies (CoDIT)*, pages 0599–0604, April 2017.

[7] P. Barnaghi, W. Wang, C. Henson, and K. Taylor. Semantics for the internet of things: Early progress and back to the future. *International Journal on Semantic Web Information Systems*, 8, 01 2012. doi: 10.4018/jswis.2012010101.

[8] M.Z. Bell. Why expert systems fail. *The Journal of the Operational Research Society*, 36(7):613–619, 1985.

[9] R.E. Bellman. *An introduction to artificial intelligence: can computers think?* San Francisco: Boyd Fraser Pub. Co, 1978.

[10] D. Brickley, R.V. Guha, and B. McBride. Rdf schema 1.1. W3c recommendation, W3C, 2014. https://www.w3.org/TR/rdf-schema.

[11] A. Bröring, S.K. Datta, and C. Bonnet. A categorization of discovery technologies for the internet of things. In *Proceedings of the 6th International Conference on the Internet of Things*, IoT16, pages 131–â139, New York, NY, USA, 2016. Association for Computing Machinery.

[12] E. Charniak and D. McDermott. *Introduction to Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985.

[13] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1. W3C Note, W3C, 2001. https://www.w3.org/TR/wsdl.html.

[14] S. Chun, S. Seo, B. Oh, and K. Lee. Semantic description, discovery and integration for the internet of things. In *Proceedings of the 2015 IEEE 9th International Conference on Semantic Computing (IEEE ICSC 2015)*, pages 272–275, Feb 2015.

[15] S. Cirani, L. Davoli, G. Ferrari, R. Léone, P. Medagliani, M. Picone, and L. Veltri. A scalable and self-configuring architecture for service discovery in the internet of things. *IEEE Internet of Things Journal*, 1 (5):508–521, 2014.

[16] L. Columbus. Internet of things (iot) intelligence update 2017. URL `https://www.forbes.com/sites/louiscolumbus/2017/11/12/2017-internet-of-things-iot-intelligence-update/#218cd6177f31`.

[17] A. Darwiche. Human-level intelligence or animal-like abilities? *CoRR*, 2017.

[18] J. Farrell and H. Lausen. Semantic annotations for wsdl and xml schema. Technical report, W3C, 2007. https://www.w3.org/TR/sawsdl/.

[19] J. Gantz and D. Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. Technical report, 2012.

[20] S. Harris, A. Seaborne, and Prud'hommeaux E. SPARQL 1.1 Query Language. W3C Recommendation, W3C, 2013. https://www.w3.org/TR/sparql11-query/.

[21] J. Haugeland. *Artificial Intelligence: The Very Idea*. Cambridge: MIT Press, 1985.

[22] H. Jo, J. Kwon, and I. Ko. Distributed service discovery in mobile iot environments using hierarchical bloom filters. In P. Cimiano, F. Frasincar, G. Houben, and D. Schwabe, editors, *Engineering the Web in the Big Data Era*, pages 498–514, Cham, 2015. Springer International Publishing.

[23] M. Klusch, P. Kapahnke, S. Schulte, F. Lecue, and A. Bernstein. Semantic web service search: A brief survey. *KI - Künstliche Intelligenz*, 30, June 2016.

[24] M. Lanthaler. Hydra core vocabulary. Technical report, Google, 2019. http://www.hydra-cg.com/spec/latest/core/.

[25] S. Legg and M. Hutter. Universal intelligence: A definition of machine intelligence. *Minds and machines*, 17(4):391–444, 2007.

[26] V. Link, Lohmann S., E. Marbach, S. Negru, and V. Wiens. Webvowl 1.1.7 [computer software], 2019. URL `http://www.visualdataweb.de/webvowl/`.

[27] M. Mahdavinejad, M. Rezvan, M. Barekatain, P. Adibi, P. Barnaghi, and A. Sheth. Machine learning for internet of things data analysis: a survey. *Digital Communications and Networks*, 4(3):161–175, 2018.

[28] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. Owl-s: Semantic markup for web services. W3c submission, W3C, 2004. https://www.w3.org/Submission/OWL-S/.

[29] P. McCorduck and C. Cfe. *Machines who think: A personal inquiry into the history and prospects of artificial intelligence*. CRC Press, 2004.

[30] E. Miller. An introduction to the resource description framework. *Bulletin of the American Society for Information Science and Technology*, 25(1):15–19, 1998.

[31] B. Motik, P.F. Patel-Schneider, and B.P. Parsia. OWL 2 Web Ontology Language. W3C Recommendation, W3C, 2012. https://www.w3.org/TR/owl2-syntax/.

[32] C. Networking. Cisco global cloud index: Forecast and methodology, 2015-2020. URL `https://www.forbes.com/sites/louiscolumbus/2017/11/12/2017-internet-of-things-iot-intelligence-update/#218cd6177f31`.

[33] N.J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[34] M. Ouzzani, L. Hendra, B. Benatallah, and A. Bouguettaya. Webfindit: An architecture and system for querying web databases. *IEEE Internet Computing*, 2(04):30–41, July 1999.

[35] F. Paganelli and D. Parlanti. A DHT-Based Discovery Service for the Internet of Things. *Journal Comp. Netw. and Communic.*, 2012:107041:1–107041:11, 2012.

[36] D. Poole, A.K. Mackworth, and R. Goebel. *Computational intelligence: A logical approach.* Oxford University Press, 2nd edition, 1998.

[37] Cambridge University Press. *Cambridge Academic Content Dictionary Reference Book*. Cambridge University Press, 2008.

[38] E. Ramos and R. Morabito. Intelligence stratum for iot. architecture requirements and functions, 2019.

[39] E. Ramos, R. Morabito, and J. Kainulainen. Distributing intelligence to the edge and beyond, 11 2018.

[40] E. Rich and K. Knight. *Artificial Intelligence*. McGraw-Hill Higher Education, 2nd edition, 1990.

[41] S.J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

[42] A.S. Tolba, A.H. El-Baz, and A.A. El-Harby. Face recognition: A literature review. *International Journal of Signal Processing*, 2(2):88–103, 2006.

[43] A. Torralba and A.A. Efros. Unbiased look at dataset bias. In *CVPR 2011*, pages 1521–1528, June 2011.

[44] W. Wang, S. De, R. Toenjes, E. Reetz, and K. Moessner. A comprehensive ontology for knowledge representation in the internet of things. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1793–1798, June 2012.

[45] Wikipedia. AI effect — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=AI%20effect&oldid=929652607`, 2020. [Online; accessed 02-March-2020].

[46] P.H. Winston. *Artificial Intelligence (2Nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

Appendix A

Example: DeepSpeech Plain English

In this appendix the complete metadata definition of the DeepSpeech Plain English service is shown in a listing. It makes use of the Intelligence Service Schema.

```
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix foaf: <http://xmlns.com/foaf.0.1/> .
@prefix iss: <http://www.ericsson.com/2019/intelligence-service-schema#> .
@prefix list: <http://jena.apache.org/ARQ/list#> .
@prefix ebu: <http://www.ebu.ch/metadata/ontologies/ebucore/ebucore#> .
@prefix gold: <http://purl.org/linguistics/gold/> .
@prefix iot: <http://www.ericsson.com/2019/iot-domain#> .
@prefix lin: <http://www.ericsson.com/2019/linguistics-domain#> .
@prefix : <http://example.com#> .

:deepspeech_plain_english a owl:NamedIndividual, iss:intelligence-service ;
                          iss:identifier "3384"^^xsd:string ;
                          iss:hasName "DeepSpeech"^^xsd:string ;
                          iss:description "DeepSpeech for personal secretary services in
      plain English"^^xsd:string ;
                          iss:hasVersion "1.3.2"^^xsd:string ;
                          iss:created "01-01-2020"^^xsd:date ;
                          iss:hasGoal iss:category-prediction ;
                          iss:hasDomain iot:speechRecognition ;
                          iss:uploadedBy :account7466 ;
                          iss:hasImplementation :NeuralNetworkImplementation .

:account7466 a owl:NamedIndividual, iss:account ;
             iss:accountName "Voice Inc"^^xsd:string ;
             iss:fromOrganization "Voice Incorporated"^^xsd:string ;
             iss:hasOwner :person8293 .

:person8293 a owl:NamedIndividual, iss:person ;
            iss:hasName "Henk-Jan Smits"^^xsd:string ;
            iss:hasReputationScore "77"^^xsd:integer .

:NeuralNetworkImplementation a owl:NamedIndividual, iss:implementation ;
                             iss:hasIdentifier "9283"^^xsd:string ;
                             iss:hasName "DeepSpeech Personal Secretary"^^xsd:string ;
                             iss:hasVersion "1.3.2"^^xsd:string ;
                             iss:hasDescription "This implementation uses a deep neural
    network."^^xsd:string ;
                             iss:created "01-01-2020"^^xsd:date ;
                             iss:hasImplementationScore "89"^^xsd:integer ;
                             iss:hasMethod iss:neural-network ;
                             iss:requires :PythonDependency ;
                             iss:requires :OnnxruntimeDepencency ;
                             iss:hasServicePipeline ( :wavcleaner :DeepSpeechOnnx ) .

```

```
:PythonDependency a owl:NamedIndividual, iss:dependency ;
                  iss:hasName "Python"^^xsd:string ;
                  iss:hasVersion "3.7.1"^^xsd:string .



:OnnxruntimeDepencency a owl:NamedIndividual, iss:dependency ;
                  iss:hasName "Onnxruntime"^^xsd:string ;
                  iss:hasVersion "5.3.2"^^xsd:string .

:wavcleaner a owl:NamedIndividual, iss:atomic-service ;
            iss:hasIdentifier "2783"^^xsd:string ;
            iss:hasName "Wav cleaner"^^xsd:string ;
            iss:hasDescription "This atomic service takes the location of an wav file as input and reduces noise in the audio."^^xsd:string ;
            iss:hasInput :WavInput .

:WavInput a owl:NamedIndividual, iss:input ;
          iss:isFile "true"^^xsd:boolean ;
          iss:hasMetaData :locatorDataTensor ;
          iss:hasDataCategory iss:audio ;
          iss:hasDomain iot:speechRecognition ;
          # the following properties are part of the iot domain, subdomain: speechRecognition
          iot:specificity "personal-secretary"^^xsd:string ;
          ebu:sampleRate "48000"^^xsd:integer ;
          ebu:hasAudioFormat "WAV"^^xsd:string ;

          iss:hasDomain lin:linguistics-domain ;
          # the following properties are part of the linguistics domain
          gold:LinguisticExpression gold:SpokenLinguisticExpression ;
          gold:Language "English"^^xsd:string ;
          gold:Dialect "Australian"^^xsd:string ;
          lin:typeOfSpeech lin:Command, lin:Question .

:locatorDataTensor a owl:NamedIndividual, iss:data-tensor ;
                   iss:elementDataType xsd:string ;
                   iss:hasDimensions ( :dimension0 ) .

:dimension0 a owl:NamedIndividual, iss:dimension ;
            iss:dimensionValue "1"^^xsd:integer .


:DeepSpeechOnnx a owl:NamedIndividual, iss:atomic-service ;
        iss:hasIdentifier "1102"^^xsd:string ;
        iss:hasName "DeepSpeech Onnx"^^xsd:string ;
        iss:hasDescription "This atomic service executes the deepspeech neural network using the onnxruntime."^^xsd:string ;
        iss:hasOutput :DeepSpeechOnnxOutput .

:DeepSpeechOnnxOutput a owl:NamedIndividual, iss:output ;
                      iss:hasDomain iss:linguistics-domain ;
                      # the following properties are part of the linguistics domain
                      gold:LinguisticExpression gold:WrittenLinguisticExpression ;

                      iss:hasDomain iss:iot-domain ;
                      # the following properties are part of the iot domain, subdomain: speechRecognition
                      iot:format iot:Language ;
                      iss:hasMetaData :outputDataTensor .

:outputDataTensor a owl:NamedIndividual, iss:data-tensor ;
                  iss:elementDataType xsd:string ;
                  iss:hasDimensions ( :dimension10 ) ;
                  iss:hasDataCategory iss:text .

:dimension10 a owl:NamedIndividual, iss:dimension ;
             iss:dimensionValue "1"^^xsd:integer .
```

Appendix B

The Intelligence Service Schema

This appendix lists the complete source code of the Intelligence Service Schema.

```
@prefix : <http://www.ericsson.com/2019/intelligence-service-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix fno: <http://www.fno.io/spec/> .
@prefix dct: <http://purl/org/dc/terms/> .
@prefix foaf: <http://xmlns.com/foaf.0.1/> .
@base <http://www.ericsson.com/2019/intelligence-service-schema> .

<http://www.ericsson.com/2019/intelligence-service-schema#> rdf:type owl:Ontology ;
  <http://purl.org/dc/terms/description> "Intelligence-Service Schema is an effort to
     develop a schema to describe intelligence services within internet of things devices
     ."@en ;
  <http://purl.org/dc/terms/hasVersion> 1.0 ;
  <http://purl.org/dc/terms/title> "Intelligence Service Schema"@en ;
  owl:imports <http://xmlns.com/foaf.0.1/> ;
  owl:imports <http://purl/org/dc/terms/> .

#################################################
# Annotation properties
#################################################

###  http://purl.org/dc/terms/description
<http://purl.org/dc/terms/description> rdf:type owl:AnnotationProperty .

###  http://purl.org/dc/terms/hasVersion
<http://purl.org/dc/terms/hasVersion> rdf:type owl:AnnotationProperty .

###  http://purl.org/dc/terms/title
<http://purl.org/dc/terms/title> rdf:type owl:AnnotationProperty .


#################################################
#   Data types
#################################################

xsd:date rdf:type rdfs:Datatype .
xsd:string rdf:type rdfs:Datatype .
xsd:integer rdf:type rdfs:Datatype .
xsd:boolean rdf:type rdfs:Datatype .

#################################################
# Data properties
#################################################

:hasIdentifier  rdf:type owl:DatatypeProperty ;
        rdfs:domain :unique-entity ;
        rdfs:range xsd:string ;
        owl:equivalentProperty dct:identifier .

```

```
:hasName  rdf:type owl:DatatypeProperty ;
      rdfs:domain :unique-entity ;
      rdfs:range xsd:string ;
      owl:equivalentProperty dct:title .

:hasVersion   rdf:type owl:DatatypeProperty ;
        rdfs:domain :unique-entity ;
        rdfs:range xsd:string ;
        owl:equivalentProperty dct:hasVersion .

:description  rdf:type owl:DatatypeProperty ;
        rdfs:domain :unique-entity ;
        rdfs:range xsd:string ;
        owl:equivalentProperty dct:description .

:created  rdf:type owl:DatatypeProperty ;
      rdfs:domain :unique-entity ;
      rdfs:range xsd:date ;
      owl:equivalentProperty dct:created .

:hasRecommendedRam  rdf:type owl:DatatypeProperty ;
          rdfs:domain :implementation ;
          rdfs:range xsd:integer .

:hasRecommendedCpuScore rdf:type owl:DatatypeProperty ;
            rdfs:domain :implementation ;
            rdfs:range xsd:integer .

:hasRecommendedStorage  rdf:type owl:DatatypeProperty ;
            rdfs:domain :implementation ;
            rdfs:range xsd:integer .

:hasRequiredRam rdf:type owl:DatatypeProperty ;
        rdfs:domain :implementation ;
        rdfs:range xsd:integer .

:hasRequiredCpuScore  rdf:type owl:DatatypeProperty ;
            rdfs:domain :implementation ;
            rdfs:range xsd:integer .

:hasRequiredStorage rdf:type owl:DatatypeProperty ;
          rdfs:domain :implementation ;
          rdfs:range xsd:integer .

:dimensionValue rdf:type owl:DatatypeProperty ;
        rdfs:domain :implementation ;
        rdfs:range xsd:integer .

:isFile rdf:type owl:DatatypeProperty ;
    rdfs:domain :input ;
    rdfs:range xsd:boolean .

:accountName  rdf:type owl:DatatypeProperty ;
        rdfs:domain :account ;
        rdfs:range xsd:string ;
        owl:equivalentProperty foaf:accountName .

:fromOrganization rdf:type owl:DatatypeProperty ;
          rdfs:domain :account ;
          rdfs:range xsd:string .

:hasReputationScore rdf:type owl:DatatypeProperty ;
          rdfs:domain :person ;
          rdfs:range xsd:integer .

:hasImplementationScore rdf:type owl:DatatypeProperty ;
            rdfs:domain :person ;
            rdfs:range xsd:integer .




##################################################
# Object properties
##################################################
:hasImplementation  rdf:type owl:ObjectProperty ;
          rdfs:domain :intelligence-service ;
          rdfs:range :implementation ;
          rdfs:comment "Connect intelligence service to an implementation."@en ;
          rdfs:isDefinedBy : ;
          rdfs:label "has implementation"@en .

:hasGoal  rdf:type owl:ObjectProperty ;
```

```
133       rdfs:domain :intelligence-service ;
134       rdfs:range :goal ;
135       rdfs:comment ""@en ;
136       rdfs:isDefinedBy : ;
137       rdfs:label "has goal"@en .
138
139 :uploadedBy   rdf:type owl:ObjectProperty ;
140         rdfs:domain :intelligence-service ;
141         rdfs:range :account ;
142         rdfs:comment "Connect an intelligence service with the account it is uploaded by
        ."@en ;
143         rdfs:isDefinedBy : ;
144         rdfs:label "uploaded by"@en .
145
146 :hasCreator   rdf:type owl:ObjectProperty ;
147         rdfs:domain :intelligence-service ;
148         rdfs:range :person ;
149         rdfs:comment "Connect an intelligence service with the person who created the
        service."@en ;
150         rdfs:isDefinedBy : ;
151         rdfs:label "has creator"@en .
152
153 :hasOwner rdf:type owl:DatatypeProperty ;
154       rdfs:domain :account ;
155       rdfs:range :person ;
156       rdfs:comment "Connect an account with the owner of the account."@en ;
157       rdfs:isDefinedBy : ;
158       rdfs:label "has owner"@en .
159
160 :usesExternalService  rdf:type owl:ObjectProperty ;
161             rdfs:domain :implementation ;
162             rdfs:range :intelligence-service ;
163             rdfs:comment "Connects an implementation with external intelligence services
        part of its service pipeline."@en ;
164             rdfs:isDefinedBy : ;
165             rdfs:label "uses external service"@en .
166
167 :hostedIn   rdf:type owl:ObjectProperty ;
168       rdfs:domain :implementation ;
169       rdfs:range :execution-environment ;
170       rdfs:comment "Connects an implementation with an execution environment in which it
        is executed."@en ;
171       rdfs:isDefinedBy : ;
172       rdfs:label "hosted in"@en .
173
174 :requires   rdf:type owl:ObjectProperty ;
175       rdfs:domain :implementation ;
176       rdfs:range :dependency ;
177       rdfs:comment "Connects an implementation with its necessary dependencies."@en ;
178       rdfs:isDefinedBy : ;
179       rdfs:label "requires"@en .
180
181 :trainedWith  rdf:type owl:ObjectProperty ;
182         rdfs:domain :atomic-service ;
183         rdfs:range :dataset ;
184         rdfs:comment "Connects an atomic service with the dataset it is trained with."
        @en ;
185         rdfs:isDefinedBy : ;
186         rdfs:label "dataset"@en .
187
188 :hasOutput  rdf:type owl:ObjectProperty ;
189       rdfs:domain :atomic-service ;
190       rdfs:range :output ;
191       rdfs:comment "Connects an atomic service with its output."@en ;
192       rdfs:isDefinedBy : ;
193       rdfs:label "has output"@en .
194
195 :hasInput   rdf:type owl:ObjectProperty ;
196       rdfs:domain :atomic-service ;
197       rdfs:range :input ;
198       rdfs:comment "Connects an atomic service with its input."@en ;
199       rdfs:isDefinedBy : ;
200       rdfs:label "has input"@en .
201
202 :hasMetadata  rdf:type owl:ObjectProperty ;
203         rdfs:domain :data-entity ;
204         rdfs:range :data-tensor ;
205         rdfs:comment "Connects input and output entities with their data tensor. The
        data tensor contains metadata about the data entity."@en ;
206         rdfs:isDefinedBy : ;
207         rdfs:label "has metadata"@en .
208
209 :hasDimensions  rdf:type owl:ObjectProperty ;
```

```
210         rdfs:domain :data-tensor ;
211         rdfs:range rdf:List ;
212         rdfs:comment "Connects a data tensor with a list of at least one dimension."@en
        ;
213         rdfs:isDefinedBy : ;
214         rdfs:label "has dimensions"@en .
215
216 :hasLabels  rdf:type owl:ObjectProperty ;
217       rdfs:domain :dataset ;
218       rdfs:range rdf:Bag ;
219       rdfs:comment "Connects a dataset with semantic annotations of its labels."@en ;
220       rdfs:isDefinedBy : ;
221       rdfs:label "has labels"@en .
222
223 :hasDomain  rdf:type owl:ObjectProperty ;
224       rdfs:domain :annotated-entity ;
225       rdfs:range :domain ;
226       rdfs:comment "Connectinon between annotated entities."@en ;
227       rdfs:isDefinedBy : ;
228       rdfs:label "has domain"@en .
229
230 :hasGoal  rdf:type owl:ObjectProperty ;
231       rdfs:domain :intelligence-service ;
232       rdfs:range :goal ;
233       rdfs:comment "Connection between an intelligence service and a goal."@en ;
234       rdfs:isDefinedBy : ;
235       rdfs:label "has goal"@en .
236
237 :hasMethod  rdf:type owl:ObjectProperty ;
238       rdfs:domain :implementation ;
239       rdfs:range :method ;
240       rdfs:comment "Connection between an implementation and a method."@en ;
241       rdfs:isDefinedBy : ;
242       rdfs:label "has method"@en .
243
244 :hasServicePipeline rdf:type owl:ObjectProperty ;
245           rdfs:domain :implementation ;
246           rdfs:range rdf:List ;
247           rdfs:comment "Connection between an implementation and a list of at least one
        atomic service."@en ;
248           rdfs:isDefinedBy : ;
249           rdfs:label "has service pipeline"@en .
250
251 :hasInputType rdf:type owl:ObjectProperty ;
252         rdfs:domain :input ;
253         rdfs:range :inputType ;
254         rdfs:comment "Connects input with a input type."@en ;
255         rdfs:isDefinedBy : ;
256         rdfs:label "has input type"@en .
257
258 :elementDataType  rdf:type owl:ObjectProperty ;
259           rdfs:domain :data-entity ;
260           rdfs:range rdf:Datatype ;
261           rdfs:comment "Descibes the datatype of the elements in the data tensor."@en .
262
263
264
265 ###############################################
266 # Classes
267 ###############################################
268
269 ###  http://www.w3.org/1999/02/22-rdf-syntax-ns#List
270 rdf:List rdf:type owl:Class .
271
272 ###  http://www.w3.org/1999/02/22-rdf-syntax-ns#Bag
273 rdf:Bag rdf:type owl:Class .
274
275 :unique-entity  rdf:type owl:Class ;
276         rdfs:comment "Unique entities, such as intelligence services, atomic services,
        dependencies, execution environments, and implementations, inherit datatype
        properties from this class."@en ;
277         rdfs:isDefinedBy : ;
278         rdfs:label "Unique Entity"@en .
279
280 :intelligence-service   rdf:type owl:Class ;
281             rdfs:comment "A declaration of an intelligence service. An intelligence
        service is a service that takes input from an application, executes a set of
        calculations and returns an output in the context of the intelligence layer. "@en ;
282             rdfs:isDefinedBy : ;
283             rdfs:label "Intelligence Service"@en ;
284             rdfs:subClassOf :unique-entity .
285
286 :account  rdf:type owl:Class ;
```

```
287       rdfs:comment "Intelligence services can be uploaded through and managed by an
          online account. The account can be owned by, and therefore represent, a person or
          company."@en ;
288       rdfs:isDefinedBy : ;
289       rdfs:label "Online Account"@en ;
290       owl:equivalentClass foaf:OnlineAccount .
291
292 :person    rdf:type owl:Class ;
293         rdfs:comment "A person. In this context a person can be the creator of an
          intelligence service or holder of an account.";
294         rdfs:isDefinedBy : ;
295         rdfs:label "person"@en ;
296         rdfs:subClassOf :unique-entity ;
297         owl:equivalentClass foaf:person .
298
299 :goal    rdf:type owl:Class ;
300       rdfs:comment "The goal of the intelligence service from a machine learning
          perspective.";
301       rdfs:isDefinedBy : ;
302     rdfs:label ""@en ;
303     rdfs:subClassOf :annotated-entity .
304
305 :method    rdf:type owl:Class ;
306       rdfs:comment "The method implemented to achieve the goal of the intelligence
          service.";
307       rdfs:isDefinedBy : ;
308       rdfs:label "Method"@en .
309
310 :domain    rdf:type owl:Class ;
311       rdfs:comment "The domain provides context in which the intelligence service
          operates. A domain taxonomy contains semantic descriptors used for input and output
          annotation."@en ;
312       rdfs:isDefinedBy : ;
313       rdfs:label "Domain"@en .
314
315 :implementation    rdf:type owl:Class ;
316           rdfs:comment "An intelligence service can have multiple implementations that
          serve the same goal. Different implementations could implement different atomic
          services, environment requirements, and data input and outputs."@en ;
317           rdfs:isDefinedBy : ;
318           rdfs:label "Implementation"@en ;
319           rdfs:subClassOf :unique-entity .
320
321 :execution-environment  rdf:type owl:Class ;
322             rdfs:comment "The execution environment contains information about the
          system that executes the code."@en ;
323             rdfs:isDefinedBy : ;
324             rdfs:label "Execution Environment"@en ;
325               rdfs:subClassOf :unique-entity .
326
327 :dependency    rdf:type owl:Class ;
328         rdfs:comment "Dependencies describe software modules necessary to execute the
          implementation."@en ;
329         rdfs:isDefinedBy : ;
330         rdfs:label "dependency"@en ;
331         rdfs:subClassOf :unique-entity .
332
333 :atomic-service    rdf:type owl:Class ;
334           rdfs:comment "A service that is directly invokable. An atomic service has no
          subprocesses and is executed in one step. A pipeline of atomic services can form an
          implementation of an intelligence service."@en ;
335           rdfs:isDefinedBy : ;
336           rdfs:label "Atomic Service"@en ;
337           rdfs:subClassOf :unique-entity .
338
339 :dataset  rdf:type owl:Class ;
340       rdfs:comment "Contains information about the dataset that is used to train the
          algorithm."@en ;
341       rdfs:isDefinedBy : ;
342       rdfs:label "Dataset"@en ;
343       rdfs:subClassOf :annotated-entity, :unique-entity .
344
345 :label  rdf:type owl:Class ;
346     rdfs:comment "Labels of categories captured by the dataset."@en ;
347     rdfs:isDefinedBy : ;
348     rdfs:label "Label"@en .
349
350 :data-entity  rdf:type owl:Class ;
351         rdfs:comment "Contains metadata about a data object. A data entity can be
          described by a domain and the semantic descriptors in the domain's taxonomy."@en ;
352         rdfs:isDefinedBy : ;
353         rdfs:label "Data-entity"@en ;
354         rdfs:subClassOf :annotated-entity .
```

```
355
356 :input  rdf:type owl:Class ;
357     rdfs:comment "Describes the input the the intelligence service requires from the
        application. An input can be described by a domain and the semantic descriptors in
        the domain's taxonomy."@en ;
358     rdfs:isDefinedBy : ;
359     rdfs:label "Input"@en ;
360     rdfs:subClassOf :data-entity .
361
362 :output   rdf:type owl:Class ;
363       rdfs:comment "Describe the output the intelligence service returns to the
        application. An output can be described by a domain and the semantic descriptors in
        the domain's taxonomy."@en ;
364       rdfs:isDefinedBy : ;
365       rdfs:label "Output"@en ;
366       rdfs:subClassOf :data-entity .
367
368 :annotated-entity rdf:type owl:Class ;
369           rdfs:comment "A dataset, goal, output, and input are subclasses of annotated
        entity. An  "@en ;
370           rdfs:inDefinedBy : ;
371           rdfs:label "Annotated entity"@en .
372
373 :data-tensor  rdf:type owl:Class ;
374         rdfs:comment "This class represents a data tensor of an input or output. A
        tensor has at least one dimension."@en ;
375         rdfs:isDefinedBy : ;
376         rdfs:label "Data Tensor"@en .
377
378 :dimension  rdf:type owl:Class ;
379       rdfs:comment "A data tensor can have has at least one dimension. Every dimension
        stores an amount of values."@wn ;
380       rdfs:isDefinedBy : ;
381       rdfs:label "Dimension"@en .
382
383 :inputType  rdfs:type owl:Class ;
384       rdfs:comment "Types of input data."@en ;
385       rdfs:isDefinedBy : ;
386       rdfs:label "input type"@en .
387
388
389 ##########################################
390 # Individuals
391 ##########################################
392
393 :audio  rdf:type :inputType, owl:NamedIndividual ;
394     rdfs:comment "Audio input."@en ;
395     rdfs:label "audio"@en .
396
397 :text   rdf:type :inputType, owl:NamedIndividual ;
398     rdfs:comment "Text input."@en ;
399     rdfs:label "Text"@en .
400
401 :image  rdf:type :inputType, owl:NamedIndividual ;
402     rdfs:comment "Image input."@en ;
403     rdfs:label "image"@en .
404
405 :tensor rdf:type :inputType, owl:NamedIndividual ;
406     rdfs:comment "Tensor input. Audio, text, image are subclasses of tensor."@en ;
407     rdfs:label "tensor"@en .
```

Listing B.1: Full RDF Turtle serialization of the intelligence service meta data model.